Decrypting the Future: A Mathematical Review of Error-Correcting Codes and Cryptography

An Honors Thesis (HONR 499)

by

Gabriela Czereszko

Thesis Advisor

Dr. Hanspeter Fischer

Ball State University

Muncie, IN

December 2017

Expected Date of Graduation

December 2017

Abstract

Data encryption plays a crucial role in postmodern society. As technology continues to advance, the current predominate system, RSA cryptography, will fail to maintain the safety of information. Thus, corporations, governments, and individuals alike must turn towards a different alternative. I discuss how the security of data is related to the mathematical properties of finite fields. Throughout the discussion, mathematical examples are demonstrated using programs in *Mathematica* code. First, error-correcting codes are presented and analyzed to demonstrate how miscommunications can effectively and efficiently be prevented. Second, the mathematical structure of ElGamal and elliptic curve cryptography is explained. Lastly, all presented cryptosystems are analyzed for their level of security and implementation.

Acknowledgments

I would like to specially thank Dr. Hanspeter Fischer for being my mentor throughout this project. Not only did he provide me with knowledge and advice, but he also stimulated my interest in cryptography and gave me the confidence to explore it.

I would also like to thank my wonderful husband and parents. They provided support and believed in me every step of the way. I would not be writing this paper if it weren't for them.

Table of Contents

Process Analysis Statement

In this paper, I explore the mathematics of finite fields and cryptography. To fully understand how computations within finite fields behave and how finite fields are structured, I first work with error-correcting codes. In this, I am able to build a working knowledge of field-generating elements, irreducible polynomials, field extensions, and order, all of which are necessary to grasp the later cryptosystem presentations. Error-correcting codes allow messages, consisting of encrypted data or plain text, to be transmitted without communication error or failure. All of the work in creating secure cryptosystems is useless if the crafted messages are received with irreversible errors. Data transmission is a direct application of mathematics to the real world. Thus, it is important to demonstrate how it can be practically implemented, and I do this in the form of *Mathematica* code. Most of the material of this section of the paper comes from weekly studies of the book *Algebraic Coding Theory* and guidance from my advisor, Hanspeter Fischer.

Once the groundwork for comprehending cryptosystems is set, I can mathematically present ElGamal and elliptic curve cryptography both through theory and programs in *Mathematica.* The programs reveal that the stated theory works as I claim that it does. Along with guidance from Hanspeter Fischer, I discovered material in *Algebraic Aspects of Cryptography* and theories from various mathematicians' published articles. Further, I draw conclusions from the research and mathematical theory and discuss how their implications prevent the world from falling victim to data breaches in the future.

I. Introduction

In a postmodern society, technology is king. Everyday tasks and transactions rely on its efficiency and security. If either of these fundamental qualities, efficiency or security, are lost, then entire portions of society's foundation will suffer. Imagine not being able to quickly and safely purchase a product online, send electronic messages, or use paperless billing. Beyond avoiding inconveniences, individuals' identities have been translated into bundles of zeros and ones, easy to work with yet easy to exploit. If a person's social security number, credit card number, or email password becomes public, they are susceptible to losing everything they had earned and saved. Therefore, it is crucial that large corporations and individuals alike ensure that their data is protected. For most of postmodern history, the standard procedure for providing such security has been through employing an RSA encryption scheme.

The RSA scheme, named after its creators Rivest, Shamir, and Adelman, exploits the mathematical property that there is no discovered algorithm to factor large numbers in a time-efficient manner [17]. Factorization, breaking down numbers into their prime divisors, is not a complex mathematical concept; however, incredible amounts of successive simple computations are not feasible to conduct in a setting where encryption and decryption are performed seemingly in an instant. The mathematical structure of the RSA cryptosystem follows.

To begin the encryption process, one party chooses two large prime numbers, p and q, and multiplies them together to create the product $n = pq$. They then compute the totient, a product defined by the formula $\phi = (p-1)(q-1)$. From there, one positive number, e, is chosen such that e is less than the totient and bears no common factors with the totient, namely

$1 \leq e < \phi$ and $GCD(e, \phi) = 1$. Once e is chosen, its multiplicative inverse modulo ϕ can be calculated call it d. In this, $ed = q\phi + 1$ for some integer ϕ. From these computations, a public and private key for the RSA system can be constructed. The set of numbers (n,e) is the public key that will be shared with any party wanting to communicate, and the set of numbers (n,d) is the private key that the original party keeps secret.

For example, suppose Noah wants to send a message M to Ella. He first needs to raise the message M to the power e mod n from Ella's public key. This needs to be done using a fast modular exponentiation algorithm called PowerMod, since in practice the exponent is several hundred digits long. For a construction of such an algorithm, see section II. The result will be the encrypted message that is sent to Ella. Once transmitted, it is up to Ella to obtain the original message using her private key. She can raise the encrypted message to the power d mod n from her private key, which will result in the correct message that Noah sent. An example of a standard transfer using *Mathematica* code is as follows.

RSA Transfer

First, define p and q as two distinct prime numbers of roughly the same size.

```
p = NextPrime[317 892 330]
q = NextPrime[432 590 000]
```

```
317 892 343
```

```
432 590 029
```

```
n = p * q;
```

```
totient = (p - 1) * (q - 1)
```

```
137 517 057 126 765 576
```

Let e be the chosen public key element that is relatively prime to the totient.

```
e = 1 398 240 093 871;
```

```
GCD[e, totient]
```

```
1
```

```
d = PowerMod[e, -1, totient]
```

```
134 700 056 018 529 031
```

Let M be the intended information message.

```
M = 938 049 830 952;
```

```
Encrypted = PowerMod[M, e, n]
```

```
18 343 125 216 612 758
```

Let Encrypted be the communicated message.

```
PowerMod[Encrypted, d, n]
```

```
938 049 830 952
```

```
M == PowerMod[Encrypted, d, n]
```

```
True
```

Thus, the inteded informaton message is equal to the decrypted message, and the transfer is successful.

The decryption is successful since e and d were designed to be multiplicative inverses of one another modulo ϕ such that $CM^d = (M^e)^d = M^{ed} = M^{q\phi+1} = M^1 = M \bmod n$. The latter follows because $x^{p-1} \equiv 1 \bmod p$ for all x that do not divide p, and likewise for q. Thus, the security of information transfers within the RSA system rests in the mathematical phenomenon that the public cannot factor the large number n to obtain the private key element d.

If a factoring algorithm is discovered, then all data that is currently protected by the RSA system could be released from its protection. Mathematicians have not yet proven if such a factoring algorithm exists. Even without a known factoring algorithm, the security of RSA has been challenged throughout its history. In recent decades, the computational speed of technology has drastically increased. According to the Federal Information Processing Standards, the modulus n should be of size 1024, 2048, or 3072 bits [9]. While having a larger modulus may appear to always be the optimal choice, larger moduli are more costly to implement because of lost time efficiency and the necessity for more powerful technology. Recently, encoders are transitioning into favoring 2048 bits over 1024 bits out of fear that the computing power of technology will soon surpass the time obstacle of factoring a number of size 1024 bits. If companies and corporations fail to update their private key alongside the advancement of technology, then their security system is breachable. The consequence of neglecting to enlarge private keys is evident in the case of the French credit card company, Carte Bleue. In 2000, 33 million credit card numbers were compromised due to the failure of a 320 bit modulus [17]. More recently, a flaw in the security chips of Estonia's national identification cards resulted in 760,000 people being banned from accessing online government services. To protect national

security, the affected people were required to update the digital information of their cards in order to regain access to the government services [21].

While breaching an RSA system through vast computational power in a "brute force" manner is possible, attacks have been designed to take advantage of possible weak points in a system's mathematical structure. I discuss a few such attacks that I have learned about from both external resources and my previous Number Theory class, MATH 416. In order to make encryption and decryption processes more time efficient, companies may choose to set the private key element d to be relatively small in value when compared to the modulus n. While increasing computational speeds as much as tenfold when using smaller exponent d can be seen as beneficial, a theorem by mathematician M. Wiener proves that the private key can be discovered if $d < \frac{1}{3}n^{\frac{1}{4}}$ where $n = pq$ and $q < p < 2q$ [4]. In other words, the increased speed of a private element satisfying a particular relationship with the modulus is at the cost of the system's security. Similarly, a party may set their public exponent e to be relatively small. While attacks in this context are not as destructive as the M. Wiener attack, the most powerful attack, attributed to mathematician Coppersmith, has the potential to quickly discover all the roots of a function f modulus n that satisfy a constraint. In particular, roots can be found when they are less than $X = n^{\frac{1}{d}-\varepsilon}$ where f is a monic polynomial with integer coefficients of degree d and ε is some non-negative integer. With that information then, the attacker has more insight into what the correct d must be [4]. More advanced attacks and hacking techniques will inevitably arise in the future given more time and computing power.

Recent advancements in quantum technology suggest that the factorization barrier will soon be breachable. What once would take an unreasonable amount of time to compute would be

executed almost instantaneously, and RSA encryption would be rendered useless. In 1994 a mathematician named Peter Shor created an algorithm that is able to efficiently factor large numbers given that quantum bits are used rather than the traditional binary digits. In binary, the bits of data have an overall value of either one or zero: 00,01,10, and 11. However, in quantum computing, the bits, termed qubits, can hold a value of both one and zero: 00+11, 00-11, 01+10 or 01-10 [7]. The simultaneous value of one and zero, termed superposition allows for two different computations to be completed at the same time. This parallel computation allows for algorithms to be executed with unmatched efficiency. The mathematician Chuang has developed a quantum system that has the ability to parallel-wise compute Shor's algorithm, and in turn, factor numbers. While fifteen is the largest number that has been successfully factored in this manner, completed with only five qubits versus the twelve required previously, the system's structure has the potential to be expanded upon [20]. It is important to note, however, that not all mathematicians believe that quantum computing will destroy RSA encryption.

A group of mathematicians working out of the University of Chicago has proposed that traditional RSA cryptography can be adapted for "post-quantum" cryptography. They suggest that quantum computations can be utilized not only to decrypt, but also to encrypt information to such a degree that the cost of decrypting it is impractical. While their conclusion is proven theoretically true, a sufficient security system in their presented model requires a modulus n that is one terabyte in size and has 4096-bit primes [2]. Even though the potential attacker is shown to have a quadratic increase in cost for decryption over the intended party, the resources required to operate with a terabyte-sized key make their statements more theoretical than applicable to real-world information security. With Chuang's recent discoveries and the recent pace of the

technology industry, it appears that the breakthrough into post-quantum computing is imminent. Hence, there needs to be another system that can encrypt information reasonably beyond the grasp of quantum computing. I discuss how the future of data security may rest in the mathematical properties of finite fields and elliptic curves. I will first explore error detection and correction in finite fields and work into the mathematics that define elliptic curve cryptography. I will present how finite field computations and alternative encryption methods can be implemented.

II. Error-Correcting Codes

Like any aspect of the real world, technology is imperfect. Not all transmitted information is received with the intended message intact; data can get lost or altered in transition. One switch of a digit can result in a drastic miscommunication or communication failure. Therefore, a successful information transfer needs protection against possible errors occurring within transmission. The mathematical properties of finite fields allow for such termed error-correcting codes. While this seems like an abstract concept, its implications are evident in real-world application. Consider how an entire barcode is not needed for an item to be scanned correctly or how QR codes are able to relay information even if they are not captured perfectly. Error-correcting codes are written in a manner that guarantees the original message is received given a certain amount of accurate data is present despite a predicted amount of missing data or errors. In general, as a code's strength of error correction increases, the transmission speed decreases. Therefore, it is up to the encoder to consider the content matter of the messages and determine the balance between the code's capacity and efficiency.

a.) Simple Binary Error-Correcting Codes

Simple binary error-correcting codes have the ability to correct substitution errors within a message. That is, they are able to identify if and where a digit has been changed. Because the only possible digits in binary codes are 1 and 0, the located erroneous digits can simply be switched to the their opposing value. Thus, even though the codes are termed "error-correcting", binary codes need only locate errors. To correct possible errors, redundancy is added into the code such that variant messages continue to resemble the intended messages enough for them to be accurately interpreted. Just as the human brain can reason through misspelled words, the addition of redundancy allows computers to make sense out of flawed information. For simple binary error-correcting codes, the redundancy is added in the form of parity check digits. Parity check digits are extra digits added to the message along with the desired data that provide computers with the information they need to make sense out of the "misspelling". The check digits are not added randomly. Rather, they are calculated using either matrix or polynomial operations as discussed in later sections.

In general, a message with n many total digits and r many check digits has $k = n - r$ many information digits. Thus, the resultant binary code has 2^k many k length codewords that can be successfully transmitted. The encoder is able to manipulate the number of possible codewords that can be generated and alter the information rate of the transmission by adjusting R when $R = \frac{k}{n}$. While increasing redundancy through the addition of check digits increases the reliability of error detection, it also decreases the information rate R, namely the proportion $\frac{k}{n} = \frac{n-r}{n}$. For example, if an encoder wants the ability to send 512 unique chunks of information, their code will require $512 = 2^9$ many unique codewords. Thus, their messages will have a

length of nine digits. Suppose that the same encoder chooses to add three check digits to instead have messages of length $n = 9 + 3 = 12$. Then, out of the possible 2^{12} messages that can be constructed, 2^9 will be codewords such that $\frac{512}{4096} = 12.5\ \%$ of the possible messages are the intended pieces of information. Encoders would like the probability that transmission errors transform one correct codeword into a different correct codeword to be low, since that phenomenon would result in a miscommunication error. In the presented example, the code is $100 - 12.5 = 87.5\%$ reliable with a transmission speed of $R = \frac{k}{n} = \frac{9}{12}$. If an encoder is unhappy with the level of reliability, redundancy can be increased by adding more check digits; however, they have to accept a lower transmission rate. If instead the encoder chooses to add six check digits, then $\frac{512}{32768} = 1.5625\ \%$ of the possible message are codewords, and the code is $100 - 1.5625 = 98.4375\%$ reliable with an information rate of $R = \frac{k}{n} = \frac{9}{15}$.

If an encoder has reason to believe that the probability of an error occurring within a specific channel is some ε such that $0 < \varepsilon < 1$, then for every 100 sent messages, ε many errors can be assumed to have occurred. Since the codes can be adapted for specified channels, simple binary error-correcting codes are highly beneficial when sending information through channels that have approximately known rates of random error occurrence. Once the desired balance between reliability and speed of transmission has been determined, the encoder is capable of constructing an appropriate code for the information.

b.) Single-Error Hamming Codes

In 1950, the mathematician Hamming published the first theory on linear codes defined by matrices [10]. Such linear codes are now termed Hamming codes. By definition, a code is linear if and only if its codewords are the set of vectors C satisfying the equation $\eta C^t = 0$ for a

matrix η. In particular, the code length n is equal to the number of columns of the matrix η [1]. Hamming's theory provided the groundwork for coding theory and quickly spurred other mathematicians to expand upon his findings. His theory states that Hamming codes have the capability of correcting all possibilities of a single error if and only if all the columns of the associated matrix are distinct and not equal to zero [1]. The associated Hamming matrix described in Berlekamp's Theorem 1.3 can be constructed by introducing a finite field of characteristic two and letting the matrix columns represent increasing powers of a field-generating element α. In this manner, the first column represents $\alpha^0 = 1 \rightarrow \{1,0,0,0\}$, and the second column represents $\alpha^1 = \alpha = \{0,1,0,0\}$.

Suppose Noah wants to send Ella a message with n digits and anticipates that one error will occur. He first needs to define a binary finite field of order n and choose a field generating element to construct the code's associated matrix. Once the matrix is set, he can determine the code's information capacity by finding a basis for the kernel of the associated matrix. The resultant dimension of the kernel is equal to the number of information digits in which Noah can form a codeword. He next needs to multiply his codeword by the transpose of the kernel matrix in order to add the check digits that will correctly generate the full message. With all n digits in place, Noah can send the message to Ella. By construction, Ella is able to run the message through the Hamming code's associated matrix via matrix multiplication to check for an error. She first computes what is called the syndrome by taking the product of the Hamming matrix and the received message. Since Noah designed the codeword to be a linear combination of the kernel basis vectors, the codeword is in the kernel of the matrix itself. Hence, if the syndrome is the zero element {0,0,0,0}, then she assumes that the received message is correct. This is where

the code's measure of reliability becomes important. If the error transformed the message into a different unintended codeword, then Ella will misinterpret Noah's message. If instead the syndrome is nonzero, then she knows an error occurred. In particular, she knows that the error is located in the position of the message that corresponds with the column of the Hamming code that matches the syndrome. This is because when the received message is ran through the coding matrix, only the error pattern will result in a non zero output. To demonstrate the communication's mathematical structure, a transfer using *Mathematica* code follows. Additional messages are considered in order to fully demonstrate the procedure. Suppose the messages are of length $n=15$, and the finite field is defined by the field extension over the finite field $\mathbb{Z}_2$ by a root of the irreducible polynomial $1+x^3+x^4$.

Consider the finite field extension over $\mathbb{Z}_2$ by a root of the irreducible polynomial $1+x^3+x^4=m(x)$

First, it is necessary to ensure that m(x) is an irreducible polynomial mod 2. All irreducibles in a field of size $2^4 - 1$ necessarily divide $x^{2^4-1}=x^{15} - 1$, and the table below shows that m(x) is one such irreducible.

```
TableForm[{Map[First, FactorList[x^15 - 1, Modulus → 2]]}]
```

$1 \quad 1+x \quad 1+x+x^2 \quad 1+x+x^4 \quad 1+x^3+x^4 \quad 1+x+x^2+x^3+x^4$

Using the finite fields package, define the chosen extension,GabField, as follows:

```
<< FiniteFields`

GabField = GF[2, {1, 0, 0, 1, 1}];
FieldIrreducible[GabField, x]
```

$1+x^3+x^4$

Let Minimalα be the minimal polynomial of the element α.

```
Minimalα = 1 + x^3 + x^4;
α = GabField[{0, 1, 0, 0}];

TableForm[{Map[First, FactorList[x^15 - 1, Modulus → 2]],
  Map[First, FactorList[x^15 - 1, Modulus → 2]] /. x → α}]
```

1	$1+x$	$1+x+x^2$	$1+x+x^4$	$1+x^3+x^4$	$1+x+x^2+x^3+x^4$
1	$\{1, 1, 0, 0\}_2$	$\{1, 1, 1, 0\}_2$	$\{0, 1, 0, 1\}_2$	0	$\{0, 1, 1, 0\}_2$

The above table demonstrates that α is in fact a root of the chosen field irreducible. Further, the table below confirms that α is a field-generating element.

```
MatrixForm[Table[α^i, {i, 0, 14}]]
```

$$\begin{pmatrix} 1 \\ \{0, 1, 0, 0\}_2 \\ \{0, 0, 1, 0\}_2 \\ \{0, 0, 0, 1\}_2 \\ \{1, 0, 0, 1\}_2 \\ \{1, 1, 0, 1\}_2 \\ \{1, 1, 1, 1\}_2 \\ \{1, 1, 1, 0\}_2 \\ \{0, 1, 1, 1\}_2 \\ \{1, 0, 1, 0\}_2 \\ \{0, 1, 0, 1\}_2 \\ \{1, 0, 1, 1\}_2 \\ \{1, 1, 0, 0\}_2 \\ \{0, 1, 1, 0\}_2 \\ \{0, 0, 1, 1\}_2 \end{pmatrix}$$

Example1 is the manual construction of the matrix from the succesive powers of α .

```
Example1 = {{1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0},
    {0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0}, {0, 0, 1, 0, 0, 0, 1, 1,
     1, 1, 0, 1, 0, 1, 1}, {0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1}};
```

Let SingleHamming be the associated Hamming matrix.

```
SingleHamming = MatrixForm[Example1]
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Define SingleHammingCode as the kernel of the Hamming matrix SingleHamming.

```
SingleHammingCode = Reverse[Mod[NullSpace[Example1], 2]];
MatrixForm[SingleHammingCode]
```

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Let M be the chosen information digits. The constructed codeword, defined Correct, is precisely the message M with the necessary 4 check digits added to the front.

```
M = {0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1};

Correct = Mod[(Transpose[SingleHammingCode].M), 2]
{1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1}
```

Suppose an error occurs during transmission such that the digit in position two of the message switches value.

```
Received = {1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1}
{1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1}

Syndrome = Mod[Example1.Received, 2]
{0, 1, 0, 0}
```

The syndrome matches the second column of SingleHamming as desired since the error was designed to have occured in the second position of the transmitted message.

Consider other messages

```
M2 = {1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1};

Correct2 = Mod[(Transpose[SingleHammingCode].M2), 2]
{1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1}

Received2 = {1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1};
```

```
Syndrome2 = Mod[Example1.Received2, 2]
{0, 1, 1, 0}
```

The syndrome matches the fourteenth column of SingleHamming as desired since the error was designed to have occured in the fourteenth position of the transmitted message.

```
M3 = {0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0};

Correct3 = Mod[(Transpose[SingleHammingCode].M3), 2]
{1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0}

Received3 = {1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0};

Syndrome3 = Mod[Example1.Received3, 2]
{1, 1, 1, 1}
```

The syndrome matches the seventh column of SingleHamming as desired since the error was designed to have occured in the seventh position of the transmitted message.

In the presented example, the syndrome matches the second column of the Hamming matrix. Hence, the error happened in position two of the received message as designed. This matching method works because matrix multiplication operates on the individual entries of the messages in successive order. Thus, a switched digit in a particular position of a word results in a change in the output matrix each time matrix multiplication operates on that position in the matrix. Notice, however, that two errors in the message would result in a decoding failure since the addition of two ones is equal to zero in binary. While Hamming codes of this form are effective at correcting a single error, they fail at correcting any additional errors.

c) Double-Error Hamming Codes

In 1960, soon after the initial publication, Hamming's findings were generalized in order to detect more than one error by mathematicians Bose, Ray-Chaudhuri, and Hocquenghem [1]. The termed BCH codes show that by employing a simple modification of the single-error correcting Hamming code, two errors can be successfully corrected. Similar to how the single error Hamming matrix represents increasing powers of field-generating element α, increasing powers of α^3 are expressed successively and added to the associated Hamming matrix beneath the rows dedicated to α. This adaptation creates a matrix that has twice the number of rows as the single error-correcting matrix, and as one may expect, the new matrix has the capability of correcting twice as many errors. It is important to note that α^3 is used instead of α^2 since the syndrome corresponding to α^{2n} is the square of the syndrome corresponding to α^n within a binary context. Just as in the case of single error-correction, the codewords are designed to be in the kernel of the code matrix. However, a nonzero syndrome of a received message now has two parts, the first and second half of the digits, $S1$ and $S3$ respectively. The mathematician

Berlekamp uses a quadratic equation, out of the calculated $S1$ and $S3$ such that the equation's roots are the reciprocal locations of the errors [1, p. 19]. While Berlekamp is accredited for constructing an efficient algorithm and formula for more than one error correction, the correction of multiple errors was proven theoretically possible by Bose and Ray-Chaudhuri in 1960. For the full mathematical argument, see reference [6]. Since computations are executed within the base field $\mathbb{Z}_2$, squaring is a linear operation. Thus, the solutions can be found through a series of matrix operations. Once the roots are calculated, their inverses match the specific columns of the code matrix where the error(s) occurred just as in the case with one error-correction.

For example, consider the same field extension as in the previous example, but now suppose that Ella receives a message from Noah with two errors instead of one. First, Ella computes the two syndromes, S1 and S3, by taking the product of the message and the code matrix and splitting it into two pieces. Next, she can plug them into Berlekamp's error-locating polynomial. By performing the appropriate linear computations required to find the polynomial's roots, taking the reciprocal of the outputs, and comparing the result with the Hamming matrix to find the matching columns, the errors can be successfully located. As before, the positions of the matching column signal the positions of the message in which errors occurred. The *Mathematica* code for the information transfer follows.

The associated Hamming Matrix for a double error-correcting matrix is constructed using both the successive powers of α and α^3 as defined below.

```
α = GabField[{0, 1, 0, 0}]
MatrixForm[Table[{α^i, (α^i)^3}, {i, 0, 14}]]
```

$\{0, 1, 0, 0\}_2$

$$\begin{pmatrix} 1 & 1 \\ \{0,1,0,0\}_2 & \{0,0,0,1\}_2 \\ \{0,0,1,0\}_2 & \{1,1,1,1\}_2 \\ \{0,0,0,1\}_2 & \{1,0,1,0\}_2 \\ \{1,0,0,1\}_2 & \{1,1,0,0\}_2 \\ \{1,1,0,1\}_2 & \{1,0,0,0\}_2 \\ \{1,1,1,1\}_2 & \{0,0,0,1\}_2 \\ \{1,1,1,0\}_2 & \{1,1,1,1\}_2 \\ \{0,1,1,1\}_2 & \{1,0,1,0\}_2 \\ \{1,0,1,0\}_2 & \{1,1,0,0\}_2 \\ \{0,1,0,1\}_2 & \{1,0,0,0\}_2 \\ \{1,0,1,1\}_2 & \{0,0,0,1\}_2 \\ \{1,1,0,0\}_2 & \{1,1,1,1\}_2 \\ \{0,1,1,0\}_2 & \{1,0,1,0\}_2 \\ \{0,0,1,1\}_2 & \{1,1,0,0\}_2 \end{pmatrix}$$

Let DoubleHamming be the associated Hamming matrix.

```
DoubleHamming = {{1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0},
    {0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0},
    {0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1}, {0, 0, 0, 1, 1, 1, 1,
     0, 1, 0, 1, 1, 0, 0, 1}, {1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1},
    {0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1}, {0, 0, 1, 1, 0, 0, 0, 1, 1,
     0, 0, 0, 1, 1, 0}, {0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0}};
```

```
MatrixForm[DoubleHamming]
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

The top four rows of DoubleHamming are the successive powers of α, and the bottom four rows are the successive powers of α^3.

```
DoubleHammingCode = Reverse[Mod[NullSpace[DoubleHamming], 2]];
```

```
MatrixForm[DoubleHammingCode]
```

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

```
MatrixForm[Transpose[DoubleHammingCode]]
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Define M to be the information digits that Noah wishes to send to Ella. The constructed codeword, defined Correct, is precisely the message M with the necessary 8 check digits added to the front.

```
M = {1, 1, 0, 0, 1, 0, 0};
Correct = Mod[Transpose[DoubleHammingCode].M, 2]
```

```
{0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0}
```

The following functions demonstrate how Noah could generate many codewords in an efficient manner.

```
Data = Partition[RandomChoice[{0, 1}, 7 * 10], 7]
```

```
{{1, 0, 0, 0, 1, 0, 0}, {0, 1, 0, 0, 0, 0, 1}, {1, 1, 1, 1, 1, 0, 1},
 {0, 0, 1, 1, 0, 0, 0}, {0, 0, 1, 1, 0, 0, 0}, {0, 1, 0, 0, 0, 0, 1},
 {1, 1, 0, 1, 0, 0, 0}, {1, 0, 0, 1, 1, 0, 0}, {0, 1, 1, 0, 1, 1, 0}, {0, 1, 0, 1, 1, 1, 1}}
```

```
CheckDigitGen = Function[x, Mod[Transpose[DoubleHammingCode].x, 2]];
```

```
Map[CheckDigitGen, Data]
```

```
{{0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0}, {1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1},
 {1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1}, {0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0},
 {0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0}, {1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1},
 {1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0}, {0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0},
 {1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0}, {0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1}}
```

Suppose Noah intended to send the codeword defined Correct, but instead Ella gets the message defined Received. Notice the two errors indicated with underlined digits.

```
Correct = {0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0};
Received = {0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0};
```

Calculating the syndrome

The syndrome of the correct codeword is the zero vector as anticipated.

```
Syndrom = Mod[DoubleHamming.Correct, 2]
```

```
{0, 0, 0, 0, 0, 0, 0, 0}
```

However, the syndrome of the received word is below.

```
Syndrom = Mod[DoubleHamming.Received, 2]
```

```
{1, 0, 0, 0, 0, 1, 0, 1}
```

The first four digits produce S1, and the second four digits produce S3:

```
S1 = GabField[{1, 0, 0, 0}];
S3 = GabField[{0, 1, 0, 1}];
```

Let Sigma be the error-locating polynomial of degree 2.

```
Sigma[z_] = 1 + S1 * z + ((S1^2) + (S3 / S1)) * z^2
```

$$1 + z\,\{1, 0, 0, 0\}_2 + z^2\,\{1, 1, 0, 1\}_2$$

The syndromes dictate the coefficients of a quadratic equation.

```
a = GabField[{1, 1, 0, 1}];
b = GabField[{1, 0, 0, 0}];
c = GabField[{1, 0, 0, 0}];
```

```
{1, α, α^2, α^3}^2
```

$$\{1, \{0, 0, 1, 0\}_2, \{1, 0, 0, 1\}_2, \{1, 1, 1, 1\}_2\}$$

The linear operations defined below solve quadratic equations within binary fields.

```
SquaringMatrix = {{1, 0, 1, 1}, {0, 0, 0, 1}, {0, 1, 0, 1}, {0, 0, 1, 1}};
MatrixForm[SquaringMatrix]
```

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

```
MatrixForm[Inverse[SquaringMatrix, Modulus → 2]];
Mod[Inverse[SquaringMatrix, Modulus → 2].{1, 0, 0, 1}, 2]
```

```
{0, 0, 1, 0}
```

The computation below checks to make sure the squaring matrix is operating properly.

```
GabField[{0, 0, 1, 0}]^2
```

$$\{1, 0, 0, 1\}_2$$

```
d = (a * c) / b^2
```

$\{1, 1, 0, 1\}_2$

```
SquareRootMatrix = Mod[SquaringMatrix + IdentityMatrix[4], 2];
Output = GabField[LinearSolve[SquareRootMatrix, {1, 1, 0, 1}, Modulus → 2]];
B1 = (b / a) Output
B2 = (b / a) (Output + 1)
```

$\{0, 1, 1, 1\}_2$

$\{0, 0, 1, 0\}_2$

Now B1 and B2 are the reciprocal locations of the errors.

```
1 / B1
```

$\{1, 1, 1, 0\}_2$

```
1 / B2
```

$\{0, 1, 1, 0\}_2$

Hence, the reciprocal locations are {1,1,1,0} and {0,1,1,0} which correspond to the eighth and fourteenth rows of DoubleHamming as designed.

While matrix operations are effective, they are not efficient when considering the time required for the encoding and decoding process. To make computations more economic, elements can be operated on as polynomials within the specified finite field rather than matrices. According to Berlekamp's Theorem 5.31, all codewords are generated by a single polynomial. In order to construct the generating polynomial, a certain product of distinct irreducible polynomials in the defined finite field is computed, call it $g(x)$, and divided out of $x^n - 1$ to obtain the polynomial $h(x) = \frac{x^n-1}{g(x)}$ where n is the number of digits in the messages. $g(x)$ necessarily divides $x^n - 1$ since the roots of the irreducibles are also roots of $x^n - 1$. By this design, any set of linearly independent multiples of h(x) constructs a code identical to the one generated from the associated Hamming matrix. The polynomial operations mimic the matrix multiplication executed in the previous examples. The coefficients of the recursion's highest degree polynomial represent the same output as the product of the chosen information and the kernel's transpose in the matrix context.

Consider the same field extension $1 + x^3 + x^4$ over $\mathbb{Z}_2$ from the previous example. Then, *g(x)* is the product of the minimal polynomials of α and α^3 such that $g(x) = (1 + x^3 + x^4)(1 + x + x^2 + x^3 + x^4) = 1 + x + x^2 + x^4 + x^8$,and $h(x) = \frac{x^{15}-1}{g(x)} = 1 + x + x^3 + x^7$. The recursive relationship between the codewords and the polynomial $h(x)$ can be programmed in *Mathematica* to produce the necessary check digits. The program generates the appropriate check digits and concatenates them with the chosen information digits at the front end of the output. The vectors formed with the additional check digits correspond with the appropriate codewords.

Not only can codeword generation be executed using polynomial operations, but the error-location process can be optimized as well. In the polynomial context, the two pieces of the syndrome, S1 and S3, are determined by the remainders from the division of α and α^3 's respective minimal polynomials out of the received message polynomial. The syndromes can then be plugged into the error-locating polynomial to successfully identify where the errors occurred as before. A *Mathematica* presentation of generating and correcting the same message and syndromes as in the matrix context but with polynomial operations follows.

Generating the Syndromes via Polynomials

The table below shows the irreducible polynomials of the field extension $x^4 + x^3+1$ over Z_2. In particular, the table reveals the minimal polynomial for α^3.

```
TableForm[{Map[First, FactorList[x^15 - 1, Modulus → 2]],
  Map[First, FactorList[x^15 - 1, Modulus → 2]] /. x → α^3}]
```

1	$1 + x$	$1 + x + x^2$	$1 + x + x^4$	$1 + x^3 + x^4$	$1 + x + x^2 + x^3 + x^4$
1	$\{1, 0, 0, 1\}_2$	$\{0, 1, 1, 0\}_2$	$\{0, 1, 0, 1\}_2$	$\{1, 1, 1, 0\}_2$	0

Let Minimalα3 be the minimal polynomial for α^3.

```
Minimalα3 = x^4 + x^3 + x^2 + x + 1;
```

Let ReceivedPoly be the polynomial representation of the message defined Received.

```
ReceivedPoly = x + x^2 + x^3 + x^4 + x^6 + x^7 + x^8 + x^9 + x^12 + x^13;
PolynomialQuotientRemainder[ReceivedPoly, Minimalα, x, Modulus → 2]
PolynomialQuotientRemainder[ReceivedPoly, Minimalα3, x, Modulus → 2]
```

$\{1 + x + x^2 + x^4 + x^9, 1\}$

$\{x + x^2 + x^4 + x^5 + x^7 + x^9, x^2 + x^3\}$

```
(α^3)^2 + (α^3)^3
```

$\{0, 1, 0, 1\}_2$

```
S1 = 1;
S3 = x + x^3;
```

Notice that the polynomials defined S1 and S3 represent the same syndromes as in the matrix context, namely {1,0,0,0} and {0,1,0,1} respectively.

Generating the Code via Polynomials

```
MatrixForm[DoubleHamming]
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

Let h0-h7 be the polynomial representations of the rows defined by DoubleHamming

```
h0 = x^14 + x^10 + x^9 + x^8 + x^7 + x^5 + x^3 + x^2;
h1 = x^13 + x^9 + x^8 + x^7 + x^6 + x^4 + x^2 + x;
h2 = x^12 + x^8 + x^7 + x^6 + x^5 + x^3 + x + 1;
h3 = x^11 + x^10 + x^9 + x^8 + x^6 + x^4 + x^3 + 1;
h4 = x^14 + x^12 + x^11 + x^10 + x^9 + x^7 + x^6 + x^5 + x^4 + x^2 + x + 1;
h5 = x^12 + x^10 + x^7 + x^5 + x^2 + 1;
h6 = x^12 + x^11 + x^7 + x^6 + x^2 + x;
h7 = x^13 + x^12 + x^8 + x^7 + x^3 + x^2;
```

Now g is the code-generating polynomial.

```
g = Minimalα * Minimalα3;
Expand[g, Modulus → 2]
```

$$1 + x + x^2 + x^4 + x^8$$

Notice that the greatest common divisor of the row polynomials h0-h7 matches the polynomial constructed by division out of $x^{15} - 1$. This is the polynomial *h*.

```
h = PolynomialGCD[h0, h1, h2, h3, h4, h5, h6, h7, Modulus → 2]
```

$$1 + x + x^3 + x^7$$

Note that the definiton of polynomials and construction of h can be automated as follows..

```
RowPolynomials = DoubleHamming.Table[x^i, {i, 14, 0, -1}]
h = PolynomialGCD @@ Join[RowPolynomials, {Modulus → 2}]
```

$$\{x^2 + x^3 + x^5 + x^7 + x^8 + x^9 + x^{10} + x^{14},\ x + x^2 + x^4 + x^6 + x^7 + x^8 + x^9 + x^{13},$$
$$1 + x + x^3 + x^5 + x^6 + x^7 + x^8 + x^{12},\ 1 + x^3 + x^4 + x^6 + x^8 + x^9 + x^{10} + x^{11},$$
$$1 + x + x^2 + x^4 + x^5 + x^6 + x^7 + x^9 + x^{10} + x^{11} + x^{12} + x^{14},\ 1 + x^2 + x^5 + x^7 + x^{10} + x^{12},$$
$$x + x^2 + x^6 + x^7 + x^{11} + x^{12},\ x^2 + x^3 + x^7 + x^8 + x^{12} + x^{13}\}$$

$$1 + x + x^3 + x^7$$

The function AddCheckDigits below recursievly computes the check digits in an economical manner. It takes the input hVector defined by all but the highest coefficient of polynomial h and computes the necessary check digits. The check digits are precisely the digits in the output up to the chosen information digits.

```
hVector = {1, 1, 0, 1, 0, 0, 0};
AddCheckDigits = Function[M, MM = M;
    For[n = 8, n > 0, n = n - 1;
     MM = Join[{Mod[Take[MM, {1, 7}].Reverse[hVector], 2]}, MM]]; MM];
AddCheckDigits[{1, 1, 0, 0, 1, 0, 0}]
```

```
{0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0}
```

Notice that the first eight digits in the output are the check digits that transform the information defined M into the codeword Correct.

```
Correct == AddCheckDigits[{1, 1, 0, 0, 1, 0, 0}]
```

```
True
```

d) *N*-Error Hamming Codes

The mathematical basis of successfully locating more than two errors is similar to the procedure for correcting two errors; however, more computations are required to find the error-locating polynomial of appropriate degree. In the example of a double error-correcting code, an error-locating polynomial of degree two was utilized to find the reciprocal locations of the errors. The degree two polynomial is just one particular example of an error-locating polynomial that can be generated by Berlekamp's algorithm for forming the error-locating polynomials of binary BCH codes. In order to correct more errors, the recursive algorithm runs for a longer duration and incorporates more variables to produce an error-locating polynomial of higher degree. In order to demonstrate the algorithm, the polynomial stated for two error-corrections in the previous example will be constructed.

First, S is defined as a polynomial in z such that the coefficients are the successive $2N\text{-}1$ many syndromes of the received message for N anticipated errors. To construct S, the syndromes for odd powers of the root α are computed using either the matrix or polynomial operations discussed previously. The even syndromes follow from the computation of the odd syndromes, since they are the squares of the preceding syndromes with half of their degree. The base for error-locating polynomials of binary codes is defined by a recursive formula with starting values $\sigma^0 = 1$ and $\tau^0 = 1$. σ^{2k} represents the error-locating polynomial of degree $2k$, and Δ_1^{2k} represents the coefficient of z^{2k+1} in the product $(1 + S)\sigma^{2k}$. The successive values of σ and τ are recursively generated as follows:

$$\sigma^{2k+2} = \sigma^{2k} + \Delta_1^{2k} z\tau^{2k}$$

$$\tau^{2k+2} = z^2\tau^{2k} \quad \textit{if } \Delta_1^{2k} = 0 \textit{ or if } \deg \sigma^{2k} > k$$

$$\tau^{2k+2} = \frac{z\sigma^{2k}}{\Delta_1^{2k}} \quad \text{if } \Delta_1^{2k} \neq 0 \text{ and deg } \sigma^{2k} \leq k$$

The recursion can be programmed in *Mathematica* to calculate the σ polynomial with the degree required to correct a specified number of errors. In general, a message with Nmany errors requires an error-locating polynomial of degree N.

For example, consider the same extension over the base field $\mathbb{Z}_2$ by a root of the irreducible polynomial $1 + x^3 + x^4$ and the same messages explored previously. The appropriate error-locating polynomial is isolated by using a function defined to express the final σ value of the recursion as a polynomial in z.

Consider the finite field of characteristic 2 with extension by irreducible polynomial $1+x^3+x^4=m(x)$.

```
α = GabField[{0, 1, 0, 0}]
```

$\{0, 1, 0, 0\}_2$

```
MatrixForm[Table[{i, α^i}, {i, 1, 14}]]
```

$$\begin{pmatrix} 1 & \{0,1,0,0\}_2 \\ 2 & \{0,0,1,0\}_2 \\ 3 & \{0,0,0,1\}_2 \\ 4 & \{1,0,0,1\}_2 \\ 5 & \{1,1,0,1\}_2 \\ 6 & \{1,1,1,1\}_2 \\ 7 & \{1,1,1,0\}_2 \\ 8 & \{0,1,1,1\}_2 \\ 9 & \{1,0,1,0\}_2 \\ 10 & \{0,1,0,1\}_2 \\ 11 & \{1,0,1,1\}_2 \\ 12 & \{1,1,0,0\}_2 \\ 13 & \{0,1,1,0\}_2 \\ 14 & \{0,0,1,1\}_2 \end{pmatrix}$$

Let OddSyndroms be defined by the syndromes calculated in the previous double error-correcting example. The power of alpha was determined by matching the sydrome with its corresponding entry in the "logarithm table" above.

```
OddSyndroms = {α^11, α^14};
S = Function[n, If[OddQ[n] == True, A = OddSyndroms[[(n + 1) / 2]], A = S[n / 2]^2];
    A];
SP = Table[S[i], {i, 1, 4}].Table[z^i, {i, 1, 4}];
1 + SP
```

Define Deg a function that determines the degree of a polynomial input.

```
Deg = Function[{pol, var}, Length[CoefficientList[pol, var]] - 1];
k = 0;
σ = 1;
τ = 1;

While[k < 2, Δ = Coefficient[(1 + SP) * σ, z^(2 k + 1)];
 PrevSigma = σ;
 σ = σ + Δ * z * τ;
 τ = If[Or[SameQ[Δ, 0], Deg[PrevSigma, z] > k], τ = z^2 * τ, τ = z * PrevSigma / Δ];
 Print["2k+2= ", 2 k + 2, ", D(2k+2)= ", Deg[PrevSigma, z],
  ", σ= ", Expand[σ], ", τ= ", Expand[τ], ", Δ= ", Δ];
 k = k + 1]
```

```
2k+2= 2, D(2k+2)= 0, σ= 1 + z {1, 0, 1, 1}2, τ= z {1, 0, 0, 1}2, Δ= {1, 0, 1, 1}2

2k+2= 4, D(2k+2)= 1, σ= 1 + z {1, 0, 1, 1}2 + z^2 {1, 1, 1, 1}2
 , τ= z {0, 1, 1, 0}2 + z^2 {1, 0, 1, 0}2, Δ= {0, 0, 1, 0}2
```

```
Expand[σ]
```

$$1 + z\,\{1, 0, 1, 1\}_2 + z^2\,\{1, 1, 1, 1\}_2$$

Define ErrorLocator to be the error-locating output σ as a polynomial in the variable x.

```
ErrorLocator = Function[x, σ /. z → x];
```

```
MatrixForm[Table[{i, ErrorLocator[1 / α^i]}, {i, 0, 14}]]
```

$$\begin{pmatrix} 0 & \{1, 1, 0, 0\}_2 \\ 1 & \{0, 1, 0, 0\}_2 \\ 2 & 0 \\ 3 & \{0, 1, 1, 1\}_2 \\ 4 & 0 \\ 5 & \{1, 1, 0, 0\}_2 \\ 6 & \{1, 1, 1, 1\}_2 \\ 7 & \{1, 1, 1, 1\}_2 \\ 8 & \{0, 1, 0, 0\}_2 \\ 9 & \{1, 0, 1, 1\}_2 \\ 10 & \{1, 0, 0, 0\}_2 \\ 11 & \{0, 0, 1, 1\}_2 \\ 12 & \{0, 1, 1, 1\}_2 \\ 13 & \{1, 0, 1, 1\}_2 \\ 14 & \{0, 0, 1, 1\}_2 \end{pmatrix}$$

The table above displays which reciprical powers of α are roots of ErrorLocator. In this, the errors occured in in the positions $\{\alpha^2, \alpha^4\}$, and the second and fourth digits of the message were switched as designed.

It is important to note that the resultant polynomial, $1 + z\{1, 0, 1, 1\}_2 + z^2\{1, 1, 1, 1\}_2$, is the same as Berlekamp's stated formula for finding two errors that is referenced in the correction of two errors as desired.

Now, suppose that a third error is introduced into the received message. Within the field extension by $1 + x^3 + x^4$, the generating polynomial for a code capable of correcting three errors is the product of the three distinct minimal polynomials of α, α^3, and α^5 respectively.

$$g(x) = (1 + x^3 + x^4)(x^4 + x^3 + x^2 + x + 1)(1 + x + x^2)$$

In general, for a code to successfully detect errors, the message must have at least as many check digits as the degree of the code generating polynomial *g(x)*. Thus, because this particular polynomial g(x) has tenth degree, ten check digits are required to successfully detect the three possible errors. In this case, a message bearing *n=15* digits has an allotment of five digits for the desired information. In fact, the calculated number of five possible chosen information digits corresponds with the dimension of the associated matrix' kernel. As presented previously, the appropriate check digits can be generated using polynomial computations and added to the information to form a codeword. In order for the error-locating algorithm to run as Berlekamp intends, the received message needs to be interpreted as coefficients of successive increasing powers of the chosen field-generating element α. As in the case of two errors, the odd-numbered syndromes are the remainders from the division of the irreducible polynomials of the respective powers of element α out of the received message polynomial. Since there are three errors being detected, σ is required to be a third degree polynomial. Thus, running the recursion algorithm program for $k < 3$ will produce the the necessary polynomial. By design, the roots of the error-locating polynomial pinpoint the reciprocal locations of the errors. By evaluating the

polynomial for each successive $\frac{1}{\alpha^i}$, $i \in \{0, 1, 2...., n-1\}$, the errors can be identified in position $\alpha^2, \alpha^4,$ and α^{12} as designed. Since the messages are written in binary language, an erroneous digit can be corrected by switching it to the opposite value. If a higher number of errors is anticipated, the recursive process is simply run for a longer loop with the appropriate value for k to account for the greater number of syndromes. The *Mathematica* program for the correction of three errors follows.

Correction of Three Errors

Consider the finite field extension over $\mathbb{Z}_2$ by a root of the irreducible polynomial 1 + x3 + x4 = m(x).

```
GabField = GF[2, {1, 0, 0, 1, 1}];
FieldIrreducible[GabField, x]
α = GabField[{0, 1, 0, 0}];
```

$1 + x^3 + x^4$

In order to correct three errors for a n=15 length message, there can only be 5 chosen information digits. This can be seen below since the constructed polynomial g has degree 10.

```
Minimalα = 1 + x^3 + x^4;
Minimalα3 = x^4 + x^3 + x^2 + x + 1;
Minimalα5 = 1 + x + x^2;
g = Minimalα * Minimalα3 * Minimalα5;
Expand[g, Modulus → 2]
PolynomialQuotientRemainder[x^15 - 1, g, x, Modulus → 2]
```

$1 + x^2 + x^5 + x^6 + x^8 + x^9 + x^{10}$

$\{1 + x^2 + x^4 + x^5, 0\}$

```
h[x_] = 1 + x^2 + x^4 + x^5;
```

```
hVector = Drop[CoefficientList[h[x], x], -1]; AddCheckDigits = Function[M, MM = M;
  For[n = 10, n > 0, n = n - 1;
   MM = Join[{Mod[Take[MM, {1, 5}].Reverse[hVector], 2]}, MM]];
  MM];
```

To communicate the message {0,1,0,1,1}, the function AddCheckDigits adds the necessary 10 check digits to the front of the message.

```
AddCheckDigits[{0, 1, 0, 1, 1}]
```

{0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1}

```
Correct = {0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1};
Received = {0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1};
```

```
ReceivedPoly = x^4 + x^6 + x^7 + x^8 + x^9 + x^11 + x^12 + x^13 + x^14;
```

```
PolynomialQuotientRemainder[ReceivedPoly, Minimalα, x, Modulus → 2] /. x -> α
PolynomialQuotientRemainder[ReceivedPoly, Minimalα3, x, Modulus → 2] /. x → (α^3)
PolynomialQuotientRemainder[ReceivedPoly, Minimalα5, x, Modulus → 2] /. x → (α^5)
```

$\{\{1, 0, 1, 0\}_2, \{0, 1, 1, 1\}_2\}$

$\{\{0, 0, 0, 1\}_2, \{1, 1, 0, 0\}_2\}$

$\{\{1, 0, 0, 0\}_2, 0\}$

```
OddSyndroms = {α^8, α^12, 0}
```

$\{\{0, 1, 1, 1\}_2, \{1, 1, 0, 0\}_2, 0\}$

```
S = Function[n, If[OddQ[n] == True, A = OddSyndroms[[(n + 1) / 2]], A = S[n / 2] ^2];
    A];
SP = Table[S[i], {i, 1, 6}].Table[z^i, {i, 1, 6}];
1 + SP
```

$1 + z^4 \{0, 0, 1, 0\}_2 + z^2 \{0, 1, 0, 0\}_2 + z \{0, 1, 1, 1\}_2 + z^6 \{1, 0, 1, 0\}_2 + z^3 \{1, 1, 0, 0\}_2$

```
Deg = Function[{pol, var}, Length[CoefficientList[pol, var]] - 1];
k = 0;
σ = 1;
τ = 1;
While[k < 3, Δ = Coefficient[(1 + SP) * σ, z^(2 k + 1)];
 PrevSigma = σ;
 σ = σ + Δ * z * τ;
 τ = If[Or[SameQ[Δ, 0], Deg[PrevSigma, z] > k], τ = z^2 * τ, τ = z * PrevSigma / Δ];
 Print["2k+2= ", 2 k + 2, ", D(2k+2)= ", Deg[PrevSigma, z],
  ", σ= ", Expand[σ], ", τ= ", Expand[τ], ", Δ= ", Δ];
 k = k + 1]
```

$2k+2= 2,\ D(2k+2)= 0,\ \sigma= 1 + z \{0, 1, 1, 1\}_2,\ \tau= z \{1, 1, 1, 0\}_2,\ \Delta= \{0, 1, 1, 1\}_2$

$2k+2= 4,\ D(2k+2)= 1,\ \sigma= 1 + z \{0, 1, 1, 1\}_2 + z^2 \{1, 1, 0, 1\}_2$
$,\ \tau= z \{0, 0, 1, 0\}_2 + z^2 \{0, 1, 0, 1\}_2,\ \Delta= \{0, 1, 1, 0\}_2$

$2k+2= 6,\ D(2k+2)= 2,\ \sigma= 1 + z^3 \{0, 0, 0, 1\}_2 + z \{0, 1, 1, 1\}_2 + z^2 \{1, 0, 0, 0\}_2$
$,\ \tau= z^2 \{1, 0, 0, 0\}_2 + z^3 \{1, 1, 0, 0\}_2 + z \{1, 1, 1, 0\}_2,\ \Delta= \{0, 1, 1, 1\}_2$

```
Expand[σ]
ErrorLocator = Function[x, σ /. z → x];
```

$1 + z^3 \{0, 0, 0, 1\}_2 + z \{0, 1, 1, 1\}_2 + z^2 \{1, 0, 0, 0\}_2$

```
MatrixForm[Table[{i, ErrorLocator[(1 / (α^i))]}, {i, 0, 14}]]
```

$$\begin{pmatrix} 0 & \{0, 1, 1, 0\}_2 \\ 1 & \{1, 0, 0, 0\}_2 \\ 2 & 0 \\ 3 & \{0, 1, 0, 1\}_2 \\ 4 & 0 \\ 5 & \{0, 1, 0, 1\}_2 \\ 6 & \{0, 0, 1, 1\}_2 \\ 7 & \{0, 1, 0, 0\}_2 \\ 8 & \{1, 0, 0, 1\}_2 \\ 9 & \{1, 0, 0, 0\}_2 \\ 10 & \{1, 0, 1, 0\}_2 \\ 11 & \{1, 0, 1, 1\}_2 \\ 12 & 0 \\ 13 & \{1, 1, 1, 0\}_2 \\ 14 & \{1, 1, 1, 1\}_2 \end{pmatrix}$$

By the table above, the errors occured in the third, fifth, and thirteenth positions of the message as designed.

e.) Alternative Method for Error-Location

While Berlekamp's algorithm for constructing an error-locating polynomial is regarded as the most efficient in its implementation, there are other known methods for error-location. One alternative method locates the errors by applying the Euclidean algorithm, a simple computational procedure [3].

Similar to Berlekamp's method, a polynomial s in the variable z is defined such that the coefficients of successive powers of z are determined by the successive syndromes of α^i for $i = 1, 2, 3, \ldots, 2e\text{-}1$ and e many errors. In particular, the constant of the polynomial begins with the syndrome of α^1. The syndromes can be found using the polynomial computations presented thus far. Once s is constructed, it is divided with remainder out of z^{2e} where e is the number of errors that are being corrected. Then, a series of divisions defined by the Euclidean algorithm is performed until the degree of the remainder, call it r_k, is less than the number of errors e. Once the algorithm is complete, the generated multipliers are stripped from the equations to form two new series of divisions. These series are equivalent to those used in finding the coefficients a,b in the equation $r_k = a \cdot z^{2e} + b \cdot s$ where r_k is the greatest common divisor of z^{2e} and s. After computing the algorithm, the final polynomial remainder corresponds with the coefficient a and is a multiple of some polynomial $u(z)$ by scalar λ. Similarly, the remainder that corresponds with b is a polynomial, call it v, that is a multiple of some polynomial $l(z)$ also by the scalar λ. Of interest is the fact that the polynomial l is proven to be exactly the desired error-locating polynomial [3]. Therefore, all that is left to identify the scalar difference λ between v and l. In fact, since l is designed to have a constant value of 1, $\lambda = l(0)$, and v need only be divided by $\lambda = l(0)$ to retrieve the desired error-locating polynomial l. Within the context of binary fields,

only the error-locating polynomial is needed since correcting errors amounts to switching a digit to its opposite value. Outside of the binary context, an error-correcting system is necessary and can be constructed using the polynomial *w*.

Consider the same message and three errors that were corrected using Berlekamp's method in the field extension by $1 + x^3 + x^4$ over the base field $\mathbb{Z}_2$. Using the already computed odd syndromes $\{\alpha^8, \alpha^{12}, 0\}$, the polynomial s is $s = \alpha^8 + \alpha z + \alpha^{12} z^2 + \alpha^2 z^3 + 0 z^4 + \alpha^9 z^5$. For the correction of $e = 3$ errors, the first quotient is $z^{2e} = z^6$. With the base set in place, the Euclidean algorithm is computed and programmed in *Mathematica*. As in the Berlekamp context, the roots of the polynomial l can be used to identify the reciprocal locations of the committed errors. While the work is not theoretically complex, it is labor-intensive and expensive to run. Even though the theory behind the mathematics of the Euclidean algorithm method is valuable in its simplicity, its ease is at the expense of its potential to be efficiently implemented.

Let s be the polynomial defined by the odd sydromes { α^8,α^{12},0}

```
s =α^8+α*z+α^12*z^2+α^2*z^3+α^9*z^5
```

$z^3 \{0, 0, 1, 0\}_2 + z \{0, 1, 0, 0\}_2 + \{0, 1, 1, 1\}_2 + z^5 \{1, 0, 1, 0\}_2 + z^2 \{1, 1, 0, 0\}_2$

Let e be the number of errors being located.

```
e = 3;
a = z^(2 e);
b = s;
Q = {};
Print["Modified Euclidean Algorithm:"]
r = b;
While[Deg[r, z] ≥ e, {q, r} = PolynomialQuotientRemainder[a, b, z];
 Q = Append[Q, q];
 Print[a, "=", q, "*", b, "+", r];
 a = b;
 b = r];
Print[]; Print["Modified Extended Euclidean Algorithm:"]
a = 0; b = 1;
While[Not[Q == {}], q = First[Q];
 r = a - q*b;
 Print[Expand[a], "=", Expand[q], "*(", Expand[b], ")+(", Expand[r], ")"];
 a = b;
 b = r;
 b = r;
 Q = Drop[Q, 1]]
Print["Up to a scalar, the error locator polynomial is:"]
Print[Expand[r]]
```

Modified Euclidean Algorithm:

$$z^6 = z\{1,1,1,1\}_2 *$$
$$z^3\{0,0,1,0\}_2 + z\{0,1,0,0\}_2 + \{0,1,1,1\}_2 + z^5\{1,0,1,0\}_2 + z^2\{1,1,0,0\}_2$$
$$+z^3\{0,0,0,1\}_2 + z\{0,0,1,1\}_2 + z^4\{0,1,1,1\}_2 + z^2\{1,1,1,0\}_2$$

$$z^3\{0,0,1,0\}_2 + z\{0,1,0,0\}_2 + \{0,1,1,1\}_2 + z^5\{1,0,1,0\}_2 + z^2\{1,1,0,0\}_2 =$$
$$z\{0,1,0,0\}_2 + \{1,0,1,1\}_2 * z^3\{0,0,0,1\}_2 + z\{0,0,1,1\}_2 + z^4\{0,1,1,1\}_2 + z^2\{1,1,1,0\}_2$$
$$+z\{0,0,0,1\}_2 + z^2\{0,1,0,1\}_2 + z^3\{0,1,1,0\}_2 + \{0,1,1,1\}_2$$

$$z^3\{0,0,0,1\}_2 + z\{0,0,1,1\}_2 + z^4\{0,1,1,1\}_2 + z^2\{1,1,1,0\}_2 = \{0,0,1,1\}_2 + z\{0,1,0,1\}_2 *$$
$$z\{0,0,0,1\}_2 + z^2\{0,1,0,1\}_2 + z^3\{0,1,1,0\}_2 + \{0,1,1,1\}_2 + z^2\{0,0,1,0\}_2 + \{1,1,1,0\}_2$$

Modified Extended Euclidean Algorithm:

$$0 = z\{1,1,1,1\}_2 * (1) + (z\{1,1,1,1\}_2)$$

$$1 = z\{0,1,0,0\}_2 + \{1,0,1,1\}_2 * (z\{1,1,1,1\}_2) + (1 + z\{0,0,1,0\}_2 + z^2\{1,1,1,0\}_2)$$

$$z\{1,1,1,1\}_2 = \{0,0,1,1\}_2 + z\{0,1,0,1\}_2 * (1 + z\{0,0,1,0\}_2 + z^2\{1,1,1,0\}_2$$
$$) + (z^3\{0,0,1,0\}_2 + \{0,0,1,1\}_2 + z^2\{0,0,1,1\}_2 + z\{1,1,1,0\}_2)$$

Up to a scalar, the error locator polynomial is:

$$z^3\{0,0,1,0\}_2 + \{0,0,1,1\}_2 + z^2\{0,0,1,1\}_2 + z\{1,1,1,0\}_2$$

Let R be the remainder of the last programmed division. R corresponds with λL where L is the desired error-locating polynomial.

```
R = z^3 GabField[{0, 0, 1, 0}] + GabField[{0, 0, 1, 1}] +
  z^2 * GabField[{0, 0, 1, 1}] + z * GabField[{1, 1, 1, 0}]
```

$$z^3\{0,0,1,0\}_2 + \{0,0,1,1\}_2 + z^2\{0,0,1,1\}_2 + z\{1,1,1,0\}_2$$

The scalar λ in this example is GabField[{0, 0, 1, 1}]$^{-1}$ since the desired error-locating polynomial is designed to have a constant coefficient of 1.

```
L = Expand[R * GabField[{0, 0, 1, 1}] ^ (-1)]
```

$$z^3\{0,0,0,1\}_2 + z\{0,1,1,1\}_2 + \{1,0,0,0\}_2 + z^2\{1,0,0,0\}_2$$

```
R = z^3 GabField[{0, 0, 1, 0}] + GabField[{0, 0, 1, 1}] +
  z^2 * GabField[{0, 0, 1, 1}] + z * GabField[{1, 1, 1, 0}]
```

$$z^3\{0,0,1,0\}_2 + \{0,0,1,1\}_2 + z^2\{0,0,1,1\}_2 + z\{1,1,1,0\}_2$$

```
L = Expand[R * GabField[{0, 0, 1, 1}] ^ (-1)]
```

$$z^3\{0,0,0,1\}_2 + z\{0,1,1,1\}_2 + \{1,0,0,0\}_2 + z^2\{1,0,0,0\}_2$$

f.) Analysis of Code Properties

Ideally, codes have a relatively high correction rate. In other words, the number of errors anticipated before coding failure is a relatively high proportion of the total number of message digits. In the given example of three error-corrections, the code has an error-correction rate of $\frac{3}{15} = 20\%$ and an information rate $\frac{5}{15} = 33.3\%$. If a code with longer message lengths is used for the same amount of error-corrections, the greater information rate is at the cost of a lower correction rate. For example, a code with messages of digit length 31 designed to correct three errors has an information rate of $\frac{16}{31} = 51.6\%$, but it has an error correction rate of $\frac{3}{31} = 9.68\%$. Along with the balance of reliability and redundancy, encoders must take the information and correct rates into account when designing their optimal code. Thus, it is important to know if all field extensions of a particular size produce a code with identical error-correction rates.

To analyze error-correction rates, first consider all field extensions over $\mathbb{Z}_2$ by a fourth degree primitive irreducible polynomial such that the field has order $2^4 - 1 = 15$. Berlekamp proves that all roots of the irreducible polynomial can be represented by successive squares of the root α. In this particular example, the roots are $\alpha, \alpha^2, \alpha^4$, and α^8, because $\alpha^{16} = \alpha$. Further then, the roots of the irreducible minimal polynomial for α^3 are the respective successive squares $\alpha^3, \alpha^6, \alpha^{12}$, and $\alpha^{24} = \alpha^9$, because $\alpha^{48} = \alpha^3$. The successive process continues until all powers of α are exhausted, and in turn, all possible irreducible polynomials of the field extension are exhausted. This partition of powers of primitive element α into roots of irreducible polynomials holds for field extensions of all orders. Consider the visual representation of the partition:

Minimal Polynomial	Field Size $2^4 - 1 = 15$	Field Size $2^5 - 1 = 31$	Field Size $2^6 - 1 = 63$
α	1,2,4,8	1,2,4,8,16	1,2,4,8,16,32
α^3	3,6,12,9	3,6,12,24,17	3,6,12,24,48,33
α^5	5,10	5,10,20,9,18	5,10,20,40,17,34
α^7	7,14,13,11	7,14,28,25,19	7,14,28,56,49,35
α^9			9,18,36
α^{11}		11,22,13,26,21	11,22,44,25,50,37
α^{13}			13,26,52,41,19,38
α^{15}	0	15,30,29,27,23	15,30,60,57,51,39
α^{17}			
α^{19}			
α^{21}			21,42
.		.	.
.		.	.
.		.	.

A gap in the middle of the partition table for minimal polynomial α^i symbolizes that the minimal polynomial of α^i is equivalent to a lower power of α^i's minimal polynomial. Hence, correcting an additional error in this case does not increase the degree of the generating polynomial $g(x)$, and thus the number of required check digits does not increase. For example, when coding messages of size $2^6 - 1 = 63$, the same number of check digits, namely 45 digits, is required to correct 8,9, and 10 errors since the irreducible polynomial for α^{17} is the same as that for α^5, and the irreducible polynomial for α^{19} is the same as that for α^{13}. Once, all of the irreducible polynomials have been accounted for in the groupings, the code is not able to correct any additional orders. The limit of error correction can be visualized in the table at the point where the irreducible polynomial has root $\alpha^n = \alpha^0 = 1$.

The roots of the distinct minimal polynomials group into particular sets of successive squares despite there being a specific choice of primitive irreducible polynomial for the field extension. The grouping dictates the required number of check digits for any *N*-error correcting code of a given length of this form. For example, to find out how many check digits are needed to correct seven errors in a message of length $2^6 - 1 = 63$, it suffices to determine which of the above groups contain the necessary powers of α, namely$\{1,3,5,7,9,11,13\}$, and add the number of total elements in those respective groupings: $6 + 6 + 6 + 6 + 3 + 6 + 6 = 39$ check digits. In this, the code has an information rate of $\frac{24}{63} = 38\%$ and a correction rate of $\frac{7}{63} = 11\%$.

Since the correction rates are determined regardless of the choice of primitive irreducible, all field extensions of a particular size produce a code with the same error-correction rate. It is important, however, to note that not all irreducible polynomials have a primitive element α that generates the field. Therefore, an encoder needs to ensure that the chosen irreducible contains a primitive element before generating a code from its field extension. In general, a field of order n has $\phi(n)$ many primitive roots where ϕ is the Euler Phi function. In the case of the field of size $2^6 - 1$, there are $\phi(63) = 36$ many primitive elements. Now, 2 has order 6 modulo 63, and further, $ORD_{63}(2) = 6$ represents the number of successive squares of a chosen root. Thus, the 36 primitive elements must be divided between $\frac{36}{6} = 6$ many irreducible polynomials. Therefore, of all possible irreducible polynomial extensions of sixth degree, there are only six possible choices of primitive codes.

While codes can still be generated with a non-primitive element, efficiency is lost. Consider the example of a code defined by a field extension over the base field $\mathbb{Z}_2$ by the irreducible polynomial $1 + x^2 + x^4 + x^5 + x^6$. Because α has order 21 and is not a primitive

element, all computations must be done modulo 21 rather than 63. Even though the resultant non-primitive code has an information rate of $\frac{12}{21} = .57$ that is comparable to the information rate of primitive degree six extensions, $\frac{30}{63} = .48$, correcting six errors in each block of 63 digits is more optimal than correcting two errors in each block of 21 digits, because the six possible errors could have been committed in any position of the larger block. A *Mathematica* demonstration of this particular non-primitive code follows.

A non-primitive double-error correcting code of length 21

The cyclotomic binary polynomial Q^n is the polynomial with coefficients in $\mathbb{Z}_2$ whose roots are precisely the elements of the algebraic closure of $\mathbb{Z}_2$ that have multiplicative order n. In this, the degree of Q^nequals EulerPhi[n]. If α is one such root, then the degree of the irreducible polynomial for α equals the multiplicative order of 2 mod n. (See Theorem 4.410 in [1]) For example, if n=21, EulerPhi[n]=2*6=12. The order of 2 mod n equals 6:

```
EulerPhi[21]
```

```
12
```

```
Table[Mod[2^i, 21], {i, 1, 6}]
```

```
{2, 4, 8, 16, 11, 1}
```

Therefore, Q^{21} factors into two irreducible polynomials of degree 6. One of them is $1 + x^2 + x^4 + x^5 + x^6$.

Start with the irreducible factor $1 + x^2 + x^4 + x^5 + x^6$ of Q^n

```
MyField = GF[2, {1, 0, 1, 0, 1, 1, 1}]
FieldIrreducible[MyField, x]
α = MyField[{0, 1, 0, 0, 0, 0, 0}]
m1[x_] = 1 + x^2 + x^4 + x^5 + x^6
m1[α]
```

```
GF[2, {1, 0, 1, 0, 1, 1, 1}]
```

$$1 + x^2 + x^4 + x^5 + x^6$$

$$\{0, 1, 0, 0, 0, 0\}_2$$

$$1 + x^2 + x^4 + x^5 + x^6$$

```
0
```

The element α has order 21

```
TableForm[Table[α^i, {i, 1, 21}]]
```

$\{0, 1, 0, 0, 0, 0\}_2$
$\{0, 0, 1, 0, 0, 0\}_2$
$\{0, 0, 0, 1, 0, 0\}_2$
$\{0, 0, 0, 0, 1, 0\}_2$
$\{0, 0, 0, 0, 0, 1\}_2$
$\{1, 0, 1, 0, 1, 1\}_2$
$\{1, 1, 1, 1, 1, 0\}_2$
$\{0, 1, 1, 1, 1, 1\}_2$
$\{1, 0, 0, 1, 0, 0\}_2$
$\{0, 1, 0, 0, 1, 0\}_2$
$\{0, 0, 1, 0, 0, 1\}_2$
$\{1, 0, 1, 1, 1, 1\}_2$
$\{1, 1, 1, 1, 0, 0\}_2$
$\{0, 1, 1, 1, 1, 0\}_2$
$\{0, 0, 1, 1, 1, 1\}_2$
$\{1, 0, 1, 1, 0, 0\}_2$
$\{0, 1, 0, 1, 1, 0\}_2$
$\{0, 0, 1, 0, 1, 1\}_2$
$\{1, 0, 1, 1, 1, 0\}_2$
$\{0, 1, 0, 1, 1, 1\}_2$
$\{1, 0, 0, 0, 0, 0\}_2$

```
n = 21
```

21

```
m = 6
```

6

```
2^m - 1
```

63

Since $n \neq 2^m-1$, the element α is not a generator for the multiplicative group of the field $\mathbb{Z}_2(\alpha)$. So, the code below will not be a primitive code.

Find the irreducible polynomial for α^3

Since every power of α is a root of x^n-1, the irreducible polynomial for α^3 divides x^n-1. It can be found by trial and error:

```
TableForm[Transpose[{Map[First, FactorList[x^21 - 1, Modulus → 2]],
   Map[First, FactorList[x^21 - 1, Modulus → 2]] /. x → α^3}]]
```

1	1
$1+x$	$\{1, 0, 0, 1, 0, 0\}_2$
$1+x+x^2$	$\{0, 0, 1, 1, 1, 1\}_2$
$1+x+x^3$	0
$1+x^2+x^3$	$\{1, 0, 1, 1, 1, 1\}_2$
$1+x+x^2+x^4+x^6$	$\{1, 0, 1, 0, 1, 1\}_2$
$1+x^2+x^4+x^5+x^6$	$\{1, 0, 0, 0, 0, 0\}_2$

```
m3[x_] = 1 + x + x^3
m3[α^3]
```

$1+x+x^3$

0

The generating polynomial

```
g[x_] = Expand[m1[x] * m3[x]]
```

$1+x+x^2+2x^3+x^4+3x^5+2x^6+2x^7+x^8+x^9$

Encoding

```
PolynomialQuotientRemainder[(x^21 - 1), g[x], x, Modulus → 2]
```

$\{1+x+x^3+x^5+x^9+x^{10}+x^{11}+x^{12}, 0\}$

```
h[x_] = 1 + x + x^3 + x^5 + x^9 + x^10 + x^11 + x^12
```

$1+x+x^3+x^5+x^9+x^{10}+x^{11}+x^{12}$

```
hVector = Drop[CoefficientList[h[x], x], -1];
AddCheckDigits = Function[M, MM = M;
  For[n = 9, n > 0, n = n - 1;
   MM = Join[{Mod[Take[MM, {1, 12}].Reverse[hVector], 2]}, MM]];
  MM];
```

Ask the random generator to create some sample message:

```
RandomInteger[1, 12]
```

$\{1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0\}$

```
M = {1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0}
```

$\{1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0\}$

Then, add the check digits:

```
Correct = AddCheckDigits[M]
```

```
{1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0}
```

Channel error

Now, create a channel error in two random places. The error-locations are underlined below:

```
Received = {1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0}
```

```
{1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0}
```

```
RR = Received.Table[x^i, {i, 0, 20}]
R = Function[z, RR /. x → z];
```

$$1 + x^3 + x^4 + x^5 + x^7 + x^8 + x^9 + x^{10} + x^{11} + x^{12} + x^{14} + x^{16} + x^{18} + x^{19}$$

Decoding using Berlekamp's algorithm

```
OddSyndroms = {PolynomialRemainder[R[x], m1[x], x, Modulus → 2] /. x → α,
  PolynomialRemainder[R[x], m3[x], x, Modulus → 2] /. x → α^3}
```

$$\{\{0, 0, 0, 0, 0, 1\}_2, \{0, 0, 0, 1, 0, 0\}_2\}$$

```
S = Function[n, If[OddQ[n] == True, A = OddSyndroms[[(n + 1) / 2]], A = S[n / 2]^2];
   A];
SP = Table[S[i], {i, 1, 4}].Table[z^i, {i, 1, 4}];
1 + SP
```

$$1 + z\{0, 0, 0, 0, 0, 1\}_2 + z^3\{0, 0, 0, 1, 0, 0\}_2 + z^2\{0, 1, 0, 0, 1, 0\}_2 + z^4\{0, 1, 0, 1, 1, 1\}_2$$

```
Deg = Function[{pol, var}, Length[CoefficientList[pol, var]] - 1];
k = 0;
σ = 1;
τ = 1;
While[k < 2, Δ = Coefficient[(1 + SP) * σ, z^(2 k + 1)];
 PrevSigma = σ;
 σ = σ + Δ * z * τ;
 τ = If[Or[SameQ[Δ, 0], Deg[PrevSigma, z] > k], τ = z^2 * τ, τ = z * PrevSigma / Δ];
 Print["2k+2= ", 2 k + 2, ", D(2k+2)= ", Deg[PrevSigma, z],
  ", σ= ", Expand[σ], ", τ= ", Expand[τ], ", Δ= ", Δ];
 k = k + 1]
```

$$2k+2= 2,\ D(2k+2)= 0,\ \sigma= 1 + z\{0, 0, 0, 0, 0, 1\}_2$$
$$,\ \tau= z\{1, 0, 1, 1, 0, 0\}_2,\ \Delta= \{0, 0, 0, 0, 0, 1\}_2$$

$$2k+2= 4,\ D(2k+2)= 1,\ \sigma= 1 + z\{0, 0, 0, 0, 0, 1\}_2 + z^2\{1, 1, 1, 1, 0, 0\}_2$$
$$,\ \tau= z\{0, 0, 0, 1, 0, 0\}_2 + z^2\{0, 1, 1, 1, 1, 1\}_2,\ \Delta= \{0, 0, 1, 0, 1, 1\}_2$$

```
ErrorLocator = Function[x, σ /. z → x];
Expand[σ]
```

$$1 + z\{0, 0, 0, 0, 0, 1\}_2 + z^2\{1, 1, 1, 1, 0, 0\}_2$$

```
MatrixForm[Table[{i, ErrorLocator[(1 / (α^i))]}, {i, 0, 20}]]
```

0	$\{0, 1, 1, 1, 0, 1\}_2$
1	$\{1, 0, 1, 0, 1, 1\}_2$
2	0
3	$\{0, 1, 0, 1, 1, 0\}_2$
4	$\{1, 1, 0, 0, 0, 1\}_2$
5	$\{0, 0, 0, 1, 0, 0\}_2$
6	$\{1, 0, 0, 1, 1, 1\}_2$
7	$\{0, 1, 1, 0, 0, 1\}_2$
8	$\{1, 0, 0, 0, 0, 0\}_2$
9	$\{0, 1, 1, 0, 1, 0\}_2$
10	$\{0, 1, 0, 0, 1, 0\}_2$
11	0
12	$\{1, 0, 1, 1, 0, 0\}_2$
13	$\{0, 0, 0, 0, 1, 1\}_2$
14	$\{1, 0, 0, 1, 0, 0\}_2$
15	$\{1, 0, 1, 0, 1, 1\}_2$
16	$\{1, 1, 1, 0, 1, 0\}_2$
17	$\{1, 0, 0, 1, 0, 0\}_2$
18	$\{0, 1, 0, 0, 0, 1\}_2$
19	$\{0, 0, 1, 0, 0, 0\}_2$
20	$\{0, 0, 0, 1, 0, 0\}_2$

Comparison to primitive code

If a primitive irreducible of degree 6 is used, then it is possible to generate a 6-error-correcting code of length 2^6-1=63 with 5*6+1*3=33 check digits (see equivalence classes of powers of 2 mod 63) and 30 message digits. While $\frac{30}{63}$=0.48 and $\frac{12}{21}$=0.57 are similar information rates, correcting 6 errors in each block of 63 is far superior to correcting 2 errors in each block of 21, because the 6 errors could be anywhere.

Another interesting example of a non-primitive code is the cyclic binary code of length 23, commonly referred to as the Golay code Q23. It is designed as a double-error correcting code, generated by a primitive 23^{rd} root of unity, call it α, where α and α^3 have the same irreducible polynomial. However, by Berlekamp's Corollary 15.27, the distance between any two codewords in Q23 is at least 7 digits [1, p. 359]. Therefore, in Q23, three errors can be successfully corrected, but not with the usual fast algorithm presented earlier. For more information on the alternative algorithm, see [1, p. 361]. The generating polynomial, g, for Q23 has degree 11, leaving Q23 with an information rate of $\frac{12}{23}$. Q23 is called a perfect 3-error-correcting code of length 23, because it fits 2^{12} pairwise disjoint spheres with radius $r = 3$ centered at the codewords into a binary alphabet of size 2^{23} without wasting any potential space when viewing the code as a metric space with distance measured by the number of errors between a potential message and a codeword. In this,
$2^{12} \cdot (C(23,0) + C(23,1) + C(23,2) + C(23,3)) = 2^{23}$ where C is the binomial coefficient. While the repetition code consisting of the two codewords {0,0,0,…,0} and {1,1,1,…,1} each of length 23 is also termed perfect, it only has information rate $\frac{1}{23}$. The set of all binary words of length 23 is precisely divided into two spheres, one made up of all codewords with more zeros than ones, and the other made up of all codewords with more ones than zeros.

Another class of perfect codes are the Hamming codes presented earlier. For example, for a single-error Hamming code with messages of length 15, there are 11 message digits and 4 check digits, forming 2^{11} pairwise disjoint spheres, so that $2^{11} \cdot (C(15,0) + C(15,1)) = 2^{15}$.

Likewise, for any 3-error correcting code of length $n = 15$ with k message digits, the product $2^k \cdot (C(15,0) + C(15,1) + C(15,2) + C(15,3)) \leq 2^{15}$, since there are 2^k disjoint spheres of radius $r = 3$ to be packed into a binary alphabet of size 2^{15}. That is, $2^k \cdot 576 \leq 2^{15}$ and $k \leq 5$. This means that even though alternative methods exists for defining error-correcting codes, there is no alternative that will create a message of length $n = 15$ with more information digits than the 3-error correcting code presented using Berlekamp's method. In analyzing the different mathematical aspects of binary error-correcting codes, encoders are able to pinpoint the code that is most efficient for their goals.

III. ElGamal Cryptography

The security of an encryption scheme rests on a mathematical problem that cannot be solved or computed quickly given that access to certain secret information is denied. In the case of RSA encryption, data is protected by the inefficiency of factoring sufficiently large numbers. Similarly, the security of ElGamal encryption, created in 1984 by mathematician Taher ElGamal, rests in the inability to efficiently take a logarithm of elements in finite fields; given an element in a finite field has been raised to some large unknown power, there is no efficient way to discover the power to which the element was raised. Within the set of real numbers, this computation is comparable to taking the logarithm of a number; hence, mathematicians term this process "the discrete logarithm problem" when operating in any group G.

Before the public keys are created, the two parties must agree upon a specified group G and an element of high order $g \in G$. Each party creates their own private key by selecting one exponent, a and b respectively, such that the exponent is relatively large within the specified group. With the initial groundwork set, the mathematics behind the encryption and decryption

are simple. Encryption occurs with one exponentiation computation, and decryption occurs with two. Since the message is required to be an element of G, it is not practical to send detailed data in this manner. Instead, the information communicated through ElGamal cryptography is often a scrambling code for information rather than the information itself. Even if two channels are working to scramble and encrypt information, the information can be read as intended given the two channels are scrambling using the same scrambling standard.

For the cryptosystem to be secure, a finite field needs to be constructed. Given the base field $\mathbb{Z}_2$, an appropriate field can be achieved through extension by an irreducible polynomial of degree higher than 1,000. The computations for ElGamal cryptography require a field-generating element or an element of large order at the least. Thus, it is important to discern whether a particular element has a satisfactory order before encryption can take place. Because all nonzero and non-unit elements in a field with prime order are generators, the task of choosing an appropriate element can be avoided if the finite field is designed to have prime order. Hence, it suffices to extend the base field $\mathbb{Z}_2$ by a polynomial that has order p, where M_p is a Mersenne prime, since the resultant finite field once the zero element is discarded will have prime order $M_p = (2^p - 1)$.

For example, consider the finite field extension over the base field $\mathbb{Z}_2$ by the irreducible polynomial $1 + x^{32} + x^{521}$ such that the field has prime order $M_{521} = (2^{521} - 1)$. Say that Noah and Ella agree upon a random $g \in G$ where G is the multiplicative group of non-zero elements of the field. Let m, an element in the field G, be the desired message, and let a and b be Noah and Ella's respective private exponents. Noah and Ella both raise the element g to their private

exponent and publicize the resultant value. Therefore, the public keys become (G, g, g^a) and (G, g, g^b).

In order for Noah to encrypt his message m, he must take the public value g^b and raise it to the power of his private exponent a. It is important to understand that even though the public values g^b and g^a are denoted in exponential form, Noah and Ella have no knowledge of each other's public element's exponent. In fact, that lack of knowledge is the exact phenomenon that provides the system's security. Since general exponentiation is an expensive computation, it is beneficial to introduce fast exponentiation, a recursive manner of exponentiation. Fast exponentiation can be programmed and utilized to make the operations more time efficient. In this, Noah's encrypted message e is computed such that $e = m(g^b)^a$. Once Ella receives the encrypted message e, she can take the public value g^a, raise it to the power of her private exponent b, and compute the multiplicative inverse call it c. Now, because the computed element c is designed to be the inverse of $(g^b)^a = (g^a)^b$, multiplying the encrypted message $e = m(g^b)^a$ by c results in the product of the original message m and the identity element 1. Hence, the message is correctly communicated. A *Mathematica* presentation of an ElGamal transfer follows.

Let p be the prime p=521 to construct the Mersenne prime $M_p = 2^{521} - 1$.

Let the irreducible polynomial be $1 + x^{32} + x^{521}$ such that every nonzero non-unit element is a field-generator.

```
Field = GF[2, Join[Join[PadRight[{1}, 32], PadRight[{1}, 489]], {1}]];
FieldIrreducible[Field, x]
```

$1 + x^{32} + x^{521}$

Let M be the message information

```
M = Field[RandomChoice[{0, 1}, 521]]
```

```
{1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1,
  0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1,
  0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1,
  1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0,
  1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1,
  1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0,
  0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1,
  1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1,
  1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1,
  0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1,
  0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1,
  1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0,
  0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0,
  0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0,
  0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0,
  1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0,
  0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1,
  0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0}_2
```

Let g be the shared element, and let a and b denote the private exponents.

```
g = Field[RandomChoice[{0, 1}, 521]];
a = RandomInteger[{1, 2^521 - 1}] ;
b = RandomInteger[{1, 2^521 - 1}];
```

FastExp is programemd to make computations in the field more practical and efficient for communication.

```
FastExp = Function[{a, n}, L = IntegerDigits[n, 2];
    answer = 1;
    s = a;
    While[Not[L == {}], p = Last[L];
     L = Drop[L, -1];
     If[p == 1, answer = answer * s];
     s = s^2];
    answer];
```

```
Encrypted = M * FastExp[FastExp[g, b], a]
```

```
{1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0,
  0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0,
  0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1,
  0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0,
  1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1,
  0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1,
  0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1,
  0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0,
  1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1,
  0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1,
  0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0,
  1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0,
  1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0,
  0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0,
  0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0,
  1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1,
  1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0,
  1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0}2
```

c is the intermediate computation once an encrypted message is received.

```
c = (1 / (FastExp[(FastExp[g, a]), b]));
```

```
Decrypted = Encrypted * c
```

```
{1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1,
  0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1,
  0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1,
  1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0,
  1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1,
  1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0,
  0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1,
  1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1,
  1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1,
  0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1,
  0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1,
  1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0,
  0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0,
  0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0,
  0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0,
  1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0,
  0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1,
  0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0}2
```

```
M == Decrypted
```

```
True
```

Thus, the decryption is successful.

The success of ElGamal lies in the complexity of finite field attributes. Elliptic curve cryptography embraces this notion and generates sufficiently complicated information transfers within the ElGamal format but for a different kind of group G.

IV. Elliptic Curve Cryptography

Elliptic curve cryptography is a manifestation of the ElGamal cryptosystem. In particular, the public group G is the group defined by the set of points within a finite field that satisfy an elliptic curve equation when paired with the operation of coordinate point addition. For an elliptic curve in $\mathbb{R}^2$, coordinate point addition is defined geometrically by computing the third point where the line through two distinct chosen points (or tangent if they are equal) hits the given elliptic curve and finding its reflection about the x-axis (here, a point at infinity is added for consistency). While the geometry changes when working within a finite field, the mathematics behind the computations is comparable besides a few additional cases. For a thorough presentation of the group operation, see [19]. Now, raising an element of the group to a particular exponent n is computed by adding the point to itself n times. While the mathematical theory behind the encryption and decryption is adjusted to accommodate the chosen group's construction, the method of communication is the same as in the standard ElGamal system. Even though the concept of elliptic curve cryptography was first introduced in 1985 by Neal Koblitz and Victor S. Miller, the efficiency and application of the cryptosystem remain active fields of research as those in the data security industry attempt to prepare for post-quantum computing.

Before encryption and decryption can begin, it is necessary to select a finite field with an appropriate prime order and elliptic equation to generate a secure group G. For optimal security and efficiency, it is important to choose the elliptic equation and prime size wisely. The security

of a finite field of order p is determined by the length of its binary expansion since the messages are ultimately translated and encoded in binary language. Currently, the government recommends that all binary fields have at least m=163, and analysis has been conducted on increasing levels of security up to m=571. For example, when m=283, the binary expansion has length of 256, resulting in security for bit length n=256. At the lowest level m=163, the expansion has length 192, and messages requires a bit length $n \in$ [161-223] [9, Appendix D]. Once the prime p has been chosen, the elliptic curve equation can be defined. While there have been algorithms developed to pseudo-randomly generate sufficient curves, the generated curves may or may not foster efficient field computations. By Hasse's Theorem [13] , the number of points in the elliptic curve group is roughly the same as the prime p.Traditional elliptic curves are defined by an equation of form $y^2 = x^3 + ax + b$.

For example, consider the curve $y^2 = x^3 - 3x + 121243$ which has been designed to be efficient by mathematicians Bos et al. This curve is termed a Weierstrass curve which have been proven to "give full ECDLP[elliptic curve discrete logarithm problem] security over prime fields of a fixed bit length, while offering good practical performance" [5]. For the security level of 256 bits, the prime p is selected as a pseudo-Mersenne prime of the form $p = 2^{512} - 569$, where 569 is the smallest positive integer such that p is prime and $512 = 2 \cdot 256$ optimizes the security of the given bit length [5, p. 5]. By construction, the prime $p \equiv 3 \bmod 4$. This property is desirable since a formula exists that is capable of computing the quadratic residues of primes of form $p \equiv 3 \bmod 4$, a necessary computation to determine the points that satisfy the chosen elliptic curve equation. Once the prime is chosen, the coefficient $a = -3$ is fixed, and the value $b = 121243$ is determined to be smallest positive value such that both the group defined by the

points on the elliptic curve and the group defined by the points on the elliptic curve's quadratic twist have prime order. If the curve has form $y^2 = x^3 - 3x + b$, then its quadratic twist is defined as $y^2 = x^3 - 3x - b$. Setting b in this manner ensures heightened resistance against Pollard's ρ-method of factorization through the control of the curve's trace, which in turn measures the complexity and security of the cryptosystem. Also, since the group of points on the elliptic curve is constructed to have prime order, all points within the group must be primitive elements. Therefore, any random point within the group has an order capable of producing effective communication.

This freedom in choosing points deems groups of elliptic curves with prime order an important field of research. In 1988, Koblitz presented that the number of primes $p \leq n$ for some n such that group order of the points on a given elliptic curve E is prime is asymptotic to $C_E n/(\log n)^2$ where C_E is a constant that depends on the curve E [12]. The probability and possibility of selecting elements bearing high order in non-prime fields, either randomly or intentionally, is a vast field of research that is still open and being researched today.

With a sufficient curve in place, the communication can be safely carried out as in the ElGamal system. Given that G is defined as the points satisfying the chosen elliptic curve, a point on the curve R is chosen as the public key component such that $R=(xR,yR)$ for integers xR, yR. In order to construct the two asymmetric ElGamal keys, two private exponents are chosen. The public keys become (G, R, R^r) and (G, R, R^t) where r and t are the chosen exponents, and the respective private keys become (G, R, r) and (G, R, t). It is important to note that since M is necessarily an element of the group G itself, the communicated messages are often unscrambling codes rather than direct information just as in the original ElGamal format.

Also, while the notation appears multiplicative, all operations are defined by the arithmetic of adding points within finite fields. Consider that Noah and Ella agree upon the curve $y^2 = x^3 - 3x + 121243 \; mod \; 2^{512} - 569$. A standard transfer in *Mathematica* code follows.

Consider the finite field $\mathbb{Z}_p$ for p defined below. Define the elliptic curve equation $y^2 = x^3 - 3x + 121243$.

```
p = 2^512 - 569;
a = -3;
b = 121243;
Mod[p, 4]
```

```
3
```

The following computations demonstrate the validity of the constructed group's prime order as stated in reference [5].

```
trace = Interpreter["HexInteger"][
  "A4C35B046B187CE4B03DA712682F4239C4A974C99F832DBC31EAC0C6FBCCA86B"]
```

```
74 524 470 523 323 306 179 098 763 612 522 339 584 975 451 401 095 028 492 149 363 970 074 628 ↪
 368 491
```

```
GroupOrd = p + 1 - trace
```

```
13 407 807 929 942 597 099 574 024 998 205 846 127 479 365 820 592 393 377 723 561 443 721 764 ↪
 030 073 472 452 331 350 974 860 724 328 926 419 335 846 901 075 402 352 787 783 454 420 582 463 ↪
 574 377 715 037
```

```
PrimeQ[GroupOrd]
```

```
True
```

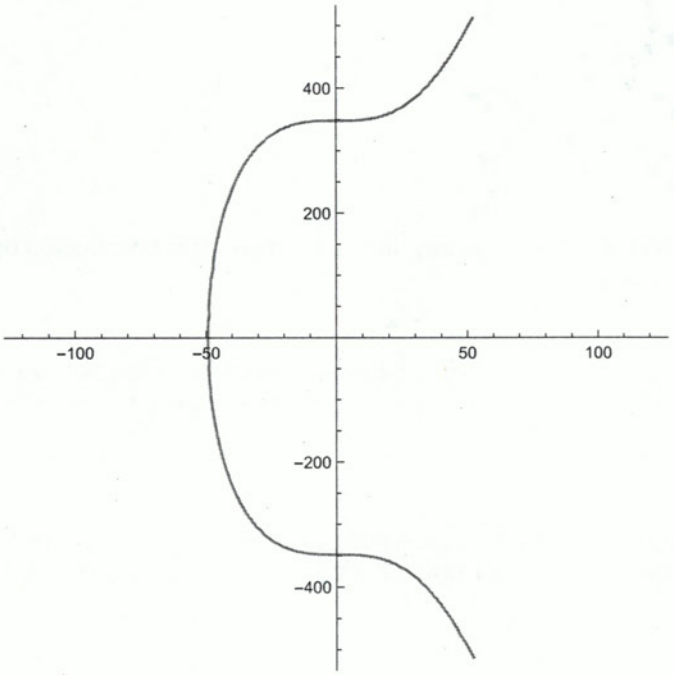
```
N[GroupOrd / p, 100]
```

```
0.99999999999999999999999999999999999999999999999999999999999999999999999999999944 ↪
 417110602476859850388 6
```

If the chosen curve was defined over the real line, it would appear as depicted on the graph below.

```
f[x_] := x^3 + a * x + b;
F[x_, y_] = f[x] - y^2
ContourPlot[F[x, y] == 0, {x, -122, 122}, {y, -512, 512}, Axes → True, Frame → False]
Solve[f[x] == 0, x]
```

$$121243 - 3x + x^3 - y^2$$



$$\left\{\left\{x \to -\left(\frac{2}{121243 - \sqrt{14699865045}}\right)^{1/3} - \left(\frac{1}{2}\left(121243 - \sqrt{14699865045}\right)\right)^{1/3}\right\},\right.$$

$$\left\{x \to \frac{1}{2}\left(1 + i\sqrt{3}\right)\left(\frac{1}{2}\left(121243 - \sqrt{14699865045}\right)\right)^{1/3} + \frac{1 - i\sqrt{3}}{2^{2/3}\left(121243 - \sqrt{14699865045}\right)^{1/3}}\right\},$$

$$\left.\left\{x \to \frac{1}{2}\left(1 - i\sqrt{3}\right)\left(\frac{1}{2}\left(121243 - \sqrt{14699865045}\right)\right)^{1/3} + \frac{1 + i\sqrt{3}}{2^{2/3}\left(121243 - \sqrt{14699865045}\right)^{1/3}}\right\}\right\}$$

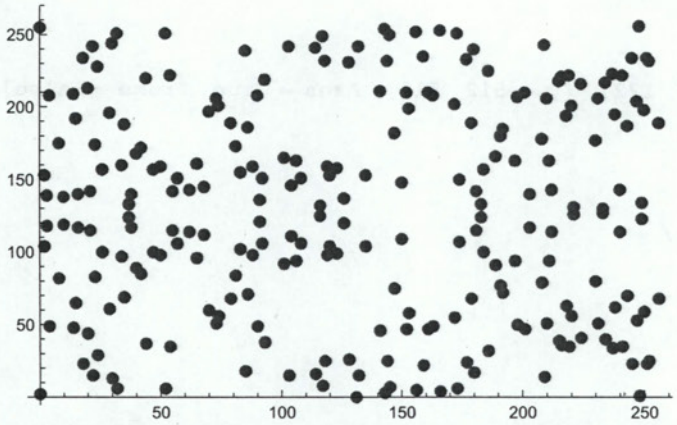
To visualize a discrete elliptic curve against the real line, consider the graph below. A different curve and field is used to present the comparison in order to have a clear depiction of points.

```
q = NextPrime[256]
f[x_] = x^3 + 5 x + 4
F[x_, y_] = y^2 - f[x];
P = Tuples[Range[q] - 1, 2];
G = Select[P, Mod[F[#[[1]], #[[2]]], q] == 0 &];
Length[G]
ListPlot[G, PlotStyle -> {Red, PointSize[Large]}]
```

257

$$4 + 5x + x^3$$

247



Let R be the shared group element for the ElGamal public key. In this example, R is constructed by adding two distinct points on the curve, P and Q.

```
xP = RandomInteger[{0, p - 1}]
```

4 518 328 551 884 945 170 266 978 977 652 761 309 679 035 611 409 195 168 783 150 456 361 824 335 ↘
097 558 423 423 725 831 410 460 583 666 049 744 244 603 012 971 109 379 440 478 079 088 549 853 ↘
591 272 040

```
yP = PowerMod[f[xP], 1 / 2, p]
```

4 746 400 305 870 961 840 775 362 325 306 215 683 406 318 755 000 227 086 592 987 147 069 445 236 ↘
019 912 511 198 351 288 015 682 040 088 546 371 765 496 657 716 230 379 030 272 862 489 301 630 ↘
044 308 843

```
xQ = RandomInteger[{0, p - 1}]
```

13 164 266 329 331 700 341 734 106 526 145 899 745 822 685 375 576 648 961 598 493 871 022 016 ↘
238 584 008 672 120 428 734 730 147 011 269 626 476 367 213 256 342 857 432 408 356 186 841 602 ↘
079 706 602 415

```
yQ = PowerMod[f[xQ], 1 / 2, p]
```

3 174 438 751 826 319 826 045 871 123 932 712 305 390 637 654 314 269 808 404 617 208 488 484 133 ↘
300 434 164 975 137 986 522 966 013 847 327 312 100 247 039 660 753 280 615 811 337 759 174 791 ↘
521 243 693

For this example, p≡3 mod 4, so the command PowerMod only has to calculate $x^{\frac{p+1}{4}}$ to find the square root of x mod p. If instead p≡1 mod 4, then an additional algorithm, while efficient, is required to compute the quadratic residue as discussed in my previous Math 416 class.

```
P = {xP, yP}
Q = {xQ, yQ}
```

```
{4 518 328 551 884 945 170 266 978 977 652 761 309 679 035 611 409 195 168 783 150 456 361 824 ↖
  335 097 558 423 423 725 831 410 460 583 666 049 744 244 603 012 971 109 379 440 478 079 088 ↖
  549 853 591 272 040,
 4 746 400 305 870 961 840 775 362 325 306 215 683 406 318 755 000 227 086 592 987 147 069 445 ↖
  236 019 912 511 198 351 288 015 682 040 088 546 371 765 496 657 716 230 379 030 272 862 489 ↖
  301 630 044 308 843}
```

```
{13 164 266 329 331 700 341 734 106 526 145 899 745 822 685 375 576 648 961 598 493 871 022 016 ↖
  238 584 008 672 120 428 734 730 147 011 269 626 476 367 213 256 342 857 432 408 356 186 841 ↖
  602 079 706 602 415,
 3 174 438 751 826 319 826 045 871 123 932 712 305 390 637 654 314 269 808 404 617 208 488 484 ↖
  133 300 434 164 975 137 986 522 966 013 847 327 312 100 247 039 660 753 280 615 811 337 759 ↖
  174 791 521 243 693}
```

The function AddingPoints executes the group operation, namely addition of points on the elliptic curve.

```
AddingPoints = Function[{P, Q}, If[SameQ[P, ∞], Answer = Q];
    If[SameQ[Q, ∞], Answer = P];
    If[And[Not[SameQ[P, ∞]], Not[SameQ[Q, ∞]]], {xP, yP} = P;
     {xQ, yQ} = Q;
     Which[Not[Or[And[xP == xQ, yP == yQ], And[xP == xQ, Mod[yP + yQ, p] == 0]]],
      s = Mod[(yP - yQ) * PowerMod[xP - xQ, -1, p], p];
      xR = Mod[(s^2 - xP - xQ), p] ;
      yR = Mod[-yP + s (xP - xR) , p];
      Answer = {xR, yR}, And[xP == xQ, Mod[yP + yQ, p] == 0], Answer = ∞,
      And[xP == xQ, yP == yQ], s = Mod[(3 * (xP^2) + a) * PowerMod[2 * yP, -1, p], p];
      xR = Mod[s^2 - 2 * xP, p];
      yR = Mod[-yP + s (xP - xR) , p];
      Answer = {xR, yR}]]; Answer];
```

```
R = AddingPoints[P, Q]
```

```
{215 026 167 209 224 793 320 023 397 797 974 308 178 721 283 099 845 538 912 390 586 695 742 984 ↖
  739 758 472 558 661 693 542 393 173 781 430 685 048 572 677 028 269 030 619 537 835 814 463 ↖
  299 660 553 072,
 5 502 461 983 872 835 852 775 428 948 904 746 180 202 618 658 404 573 510 151 548 367 616 853 ↖
  701 864 882 719 661 719 078 397 570 211 781 349 505 585 049 277 636 866 288 422 169 711 339 ↖
  000 022 967 586 738}
```

The function FastExpAdd repeatedly adds a point to itself *n*-many times.

```
FastExpAdd = Function[{P, n}, L = IntegerDigits[n, 2];
    answer = ∞;
    q = P;
    While[Not[L == {}], w = Last[L];
     L = Drop[L, -1];
     If[w == 1, answer = AddingPoints[answer, q]];
     q = AddingPoints[q, q]];
    answer];
```

Let t and r be the randomly chosen private key exponents.

```
t = RandomInteger[{1, p - 1}];
r = RandomInteger[{1, p - 1}];
```

Define Info as the desired message consisting of 100 binary information digits. Info is mapped into the x-coordinate of a point on the elliptic curve.

```
Info = {1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1,
    0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0,
    1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0,
    0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1};
```

Extra digits are added to the end of the x-coordinate binary message to satisfy the condition that the point is on the curve. Call the constructed point M.

```
xM = FromDigits[Join[Info, {1, 0, 1, 0, 0}], 2];
yM = PowerMod[f[xM], 1 / 2, p];
M = {xM, yM}
```

```
{23 650 956 938 657 134 206 802 555 216 564,
 1 722 776 345 872 171 659 858 732 879 421 104 300 376 843 567 898 554 365 362 986 607 738 793 ↘
  309 441 015 252 381 188 746 995 642 829 948 522 034 971 998 387 746 173 448 848 038 925 510 ↘
  039 161 914 690 985}
```

```
Rt = FastExpAdd[R, t];
Rr = FastExpAdd[R, r];

Encrypted = AddingPoints[M, FastExpAdd[Rt, r]]
```

```
{11 477 265 483 780 686 290 670 920 300 078 492 008 219 722 821 553 742 401 090 449 209 919 752 ↘
  816 154 161 927 976 117 864 552 035 258 497 440 520 853 137 445 420 735 874 726 177 707 291 ↘
  208 029 068 718 894,
 5 602 102 786 499 023 214 275 768 206 024 291 978 183 975 329 177 630 462 355 029 562 470 636 ↘
  125 695 244 486 428 216 416 125 633 683 535 926 504 538 390 580 114 067 423 000 394 330 176 ↘
  139 046 286 331 077}
```

```
{xRrt, yRrt} = FastExpAdd[Rr, t]
```

```
{12 496 396 271 096 868 819 751 012 005 605 618 303 477 367 419 492 374 994 268 123 220 413 565 ↘
  926 918 574 976 748 081 015 431 228 011 638 168 809 202 170 498 471 169 158 309 396 447 177 ↘
  752 046 888 676 052,
 1 902 622 412 004 097 980 777 057 697 204 298 811 541 416 743 864 727 119 197 174 454 225 852 ↘
  271 355 612 575 490 896 879 298 665 432 812 453 172 321 466 266 672 855 504 179 678 601 671 ↘
  029 156 782 787 107}
```

```
Decrypted = AddingPoints[Encrypted, {xRrt, -yRrt}]
```

```
{23 650 956 938 657 134 206 802 555 216 564,
 1 722 776 345 872 171 659 858 732 879 421 104 300 376 843 567 898 554 365 362 986 607 738 793 ↘
  309 441 015 252 381 188 746 995 642 829 948 522 034 971 998 387 746 173 448 848 038 925 510 ↘
  039 161 914 690 985}
```

```
M == Decrypted
```

```
True
```

```
Output = IntegerDigits[Part[Decrypted, 1], 2]
```

```
{1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0,
 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1,
 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0,
 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0}
```

After translating the decrypted message back into binary langauage, the first 100 digits correspond to the original 100 digits of the desired message.

```
Info == Take[Output, 100]
```

```
True
```

Thus, the decryption is successful.

V. Conclusion

As data continues to support the backbone of society, it is crucial that data is protected in the most secure yet efficient manner possible. Modern cryptography is caught in a transitional period as governments, companies, and individuals alike fight to keep up with the increasing speed of technology and prepare for a revolutionary quantum age. Researchers are adapting, innovating, and producing new ideas in order to ensure that the security of our society, so heavily rooted in data, is preserved. The data breach of Carte Bleue and the chip card ban in Estonia demonstrate this dire need. As the computational power of technology increases, the bit length of key sizes needs to increase as well. When comparing the encryption power of varying key lengths between the RSA and elliptic curve cryptosystem, it follows that elliptic curve cryptography requires smaller key sizes for parallel levels of standard security.

Even though studies done by the mathematician Okeyinka show that RSA cryptography is more energy-efficient overall, ElGamal cryptography is more efficient when only considering the decryption procedure. In particular, given a text length of 18,580 characters, decryption requires 111.9454 milliseconds and 162.4227 milliseconds for ElGamal and RSA respectively [18]. Both methods, while swelled in magnitude, will continue to operate as long as quantum technology fails to emerge. Thus, Okeyinka concludes that combining the best qualities of each system results in the most economic system yet; however, he does not take quantum technology into consideration. Unfortunately, in the realm of quantum technology, the same Shor's Algorithm that can deteriorate an RSA system also has the potential to crumble the elliptic curve cryptosystem. For a deeper look into Shor's Algorithm, refer to Lowe's article [16]. Fortunately, there are cryptosystems rooted in the foundations of ElGamal and ultimately elliptic curve

cryptography emerging on the horizon that are designed to be quantum technology resistant. The mathematician Hecht states that one particular generalization of ElGamal cryptography has the ability to maintain security in spite of quantum computing. His argument is especially appealing to cryptographers as all computations remain secure within the finite field $\mathbb{Z}_{251}$, a relatively small field within the cybersecurity context [11]. The supposed cryptosystem works within the general linear multiplicative subgroup over prime field F_{251}, denoted $GL(d, F_{251})$. where d is the order of a square matrix within the subgroup.

Researchers Li, Higgins, and Clement out of Brigham Young University compare the speed of transfers in both the original and elliptic curve version of the ElGamal method for finite fields of bit lengths 768 and 1024. They state that for parallel levels of security, the elliptic curve version of ElGamal can be executed over 50 times faster than the standard ElGamal system [15]. Further, mathematicians have developed a slight modification of the current elliptic curve method that also has the potential to withstand quantum attacks and the computing power of Shor's Algorithm. The concept is based on isogenies, rational maps between curves with equal order, of supersingular curves. Supersingular curves, by definition, have $p \pm 1$ points such that no points have order p where p is the prime that defines the finite field [22]. Since this adaptation was only first proposed in 2014, researchers are still working to determine its level of security and efficiency. While there is still much to learn, the amount of explorable properties of elliptic curves, and supersingular curves in particular, hold great potential for future cryptosystems.

As computing speeds quicken and society progresses into the quantum age of technology, cryptosystems need to evolve in order to maintain security. The current predominate encryption method, RSA cryptography, is not equipped to adapt to the changing tides of technology.

Even if the key sizes are continually increased, the cost of the necessary computing power is not practical to uphold. Corporations and individuals alike must turn towards finite fields to ensure the safety of their information. The mathematical structure of finite fields produces encryption methods that have the potential to keep data secure alongside the progression of technology. ElGamal and elliptic curve cryptosystems are both active areas of research that offer hopeful alternatives to the current RSA system. Further, the error-correcting codes that are defined within binary finite fields protect the transmission of all data, whether it be encrypted or not. Without error-correcting codes, the work put into cleverly encrypting data can be quickly undone during transmission. Beyond the protection of sensitive information, error-correcting codes are utilized for many everyday tasks such as making a cup of coffee with a Nespresso machine, scanning QR codes with cell phones, saving files on computers, and listening to CD players on bumpy car rides. Postmodern society cannot afford to lose either security or efficiency, and the mathematical properties of finite fields and elliptic curves offer insight into what society needs to keep the peace.

References

1. Berlekamp, E. R. (1968). *Algebraic Coding Theory*. New York, NY: McGraw-Hill Book Company.

2. Bernstein, D. J., Heninger, N., Lou, P., & Valenta, L. (2017). Post-quantum RSA, In: *Post-Quantum Cryptography, Lecture Notes in Computer Science,* Springer, 311-329.

3. Bierbrauer, J. (2017). *Introduction to Coding Theory* (2nd ed.). New York, NY: Taylor & Francis Group, LLC.

4. Boneh, D. (1999). Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS,* Vol 46, no.2, 203-213.

5. Bos, J. W., Costello, C., Longa, P., & Naehrig, M. (2016). Selecting elliptic curves for cryptography: an efficiency and security analysis. *Journal of Cryptographic Engineering*, Vol 6, no.4, 259-286.

6. Bose, R. C., & Ray-Chaudhuri, D. K. (1960). On a class of error correcting binary group codes. *Information and Control,* no.6, 68-79.

7. Chu, J. (2016, March 03). The beginning of the end for encryption schemes? , "MIT News" .[Retrieved October 10, 2017], http://news.mit.edu/2016/quantum-computer-end-encryption-schemes-0303

8. Cohen, H., Frey, G., & Avanzi, R. M. (2006). *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Boca Raton, FL: Taylor and Francis.

9. Digital Signature Standard (DSS). (2013). National Institute of Standards and Technology (NIST). *Federal Information Processing Standards Publication 186-4 (FIPS)*

10. Hamming, R.W. (1950). "Error Detecting and Error Correcting Codes". *Bell System Tech.* Vol 29. no.2, pp.147-160

11. Hecht, J. (2017, February). Post-Quantum Cryptography(PQC): Generalized ElGamal Cipher over GF(251^8). Preprint. At arxiv.org, arXiv:1702.03587.

12. Koblitz, N. (1988). Primality of the number of points on an elliptic curve over a finite field. *Pacific Journal of Mathematics,* Vol 131, no.1, 157-165.

13. Koblitz, N. (1994). *A Course in Number Theory and Cryptography* (2nd ed.). New York: Springer.

14. Koblitz, N. (1998). *Algebraic Aspects of Cryptography* (Vol. 3, Algorithms and Computation in Mathematics). Berlin: Springer.

15. Li, Z., Higgins, J., & Clement, M. (2001). Performance of finite field arithmetic in an elliptic curve cryptosystem. In: *MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems.*

16. Lowe, R. (2003). Breaking the code. *B.S Undergraduate Mathematics Exchange,* Vol 1, no.1,.35-39.

17. Nisha, S., & Farik, M. (2017). RSA public key cryptography algorithm – A Review. *International Journal of Scientific & Technology Research,* Vol 6, no.7, 187-191.

18. Okeyinka, AE (2015). Computational speeds analysis of RSA and ElGamal algorithms on text data. In: *Proceedings of The World Congress on Engineering and Computer Science 2015, WCECS 2015. Lecture Notes in Engineering and Computer Science,* 21–23 October, 2015, San Francisco, USA, pp.115–118

19. O'Maley, A. (2005). Elliptic curves and elliptic curve cryptography. *B.S. Undergraduate Mathematics Exchange,* Vol 3, no.1, 16-24.

20. Russon, M. (2016, March 07). MIT developing scalable quantum computer based on five atoms that could end RSA encryption. "International Business Times". [Retrieved October 10, 2017], http://www.ibtimes.co.uk/mit-developing-scalable-quantum-computer-based-five-atoms-that-could-end-rsa-encryption-1548079

21. Security flaw forces Estonia ID 'lockdown'. (2017, November 03). BBC News. [Retrieved November 18, 2017], http://www.bbc.com/news/technology-41858583

22. Wohlwend, J. (2016). Elliptic curve cryptography: pre and post quantum. Preprint. [Retrieved October, 2017], https://www.semanticscholar.org/paper/Elliptic-Curve-Cryptography-Pre-and-Post-Quantum-Wohlwend/f012708bb97b0af085b3f95c72ca2538ccf34eb6.